Android Dalvik虚拟机结构及机制剖析

——第2卷 Dalvik虚拟机各模块机制分析

吴艳霞 张国印 编著



清华大学出版社

Android Dalvik 虚拟机结构及机制剖析
——第 2 卷 Dalvik 虚拟机各模块机制分析

吴艳霞　张国印　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本系列丛书共分 2 卷，本书为第 2 卷，在第 1 卷的基础上，采有情景分析的方式对 Android Dalvik 虚拟机的源代码进行了有针对性的分析，围绕类加载、解释器、即时编译、本地方法调用、内存管理及反射机制等功能模块展开分析，主要帮助读者从微观上更深入地理解 Dalvik 虚拟机中各功能模块的实现原理及运行机制。

第 2 卷共 6 章，第 1 章介绍类加载机制，包括其整体的工作流程和机制，详细讲解了其中的三个阶段，并以一个实例验证了源码分析的结果；第 2 章介绍了 Dalvik 虚拟机中至关重要的内存管理机制，详细讲解了其实现的两种算法；第 3 章分析了 JNI 模块的实现原理，在分析源码的基础上，细致入微地介绍了为何用 JNI 编程会提升程序的执行效率；第 4 章以反射机制的一个代码示例开始，介绍了其涉及的 API，并从宏观到微观详细介绍了反射机制；第 5 章介绍了实现解释器的两种不同的技术，比较了 Fast 解释器和 Portable 解释器的不同及各自的优劣势，第 6 章从介绍最近在解释器中非常火的 JIT(即时编译)开始，到 JIT 的所谓的前端分析，再到 JIT 的后端代码生成，为本书画上一个圆满的句号。

通过阅读本书，读者可以了解 Dalvik 虚拟机在 Android 应用程序运行过程中所扮演的重要角色及其不可替代的价值。通过阅读本系列丛书，读者可以对 Android 应用程序的执行过程有更加细致的了解，可以帮助读者优化自己编写的应用程序，更加合理地设计应用程序结构，有效提高应用程序的运行速度。

图书在版编目(CIP)数据

Android Dalvik 虚拟机结构及机制剖析. 第 2 卷，Dalvik 虚拟机各模块机制分析/吴艳霞，张国印编著. --北京：清华大学出版社，2014

ISBN 978-7-302-36108-4

Ⅰ. ①A…　Ⅱ. ①吴…　②张…　Ⅲ. ①移动终端—应用程序—程序设计—虚拟处理机—研究　Ⅳ. ①TP338

中国版本图书馆 CIP 数据核字(2014)第 069718 号

责任编辑：袁勤勇　薛　阳
封面设计：
责任校对：焦丽丽
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：10.75　　**字　　数**：262 千字
版　　次：2014 年 8 月第 1 版　　**印　　次**：2014 年 8 月第1 次印刷
印　　数：1～　000
定　　价：　.00 元

产品编号：056943-01

前言

读者在看过第 1 卷之后，应该已经大致了解了 Dalvik 虚拟机，包括其模块组成、Dex 文件格式以及使用到的工具。本书作为第 2 卷，在第 1 卷的基础上，将要细致地分析 Dalvik 内部的组成原理，采用情景分析的方式，有针对性地分析 Android Dalvik 虚拟机的源代码，围绕类加载、解释器、即时编译、本地方法调用、内存管理及反射机制等功能模块逐个击破，帮助各位读者从微观上更深入地理解 Dalvik 虚拟机中各功能模块的实现原理及运行机制。在其中展示的所有源码，都来自于 Android 4.0.4 的源码，并在此基础上添加了些许注释，以便理解源码。源码篇幅可能有点长，但大多数都是其中最主要的内容，如果读者手边有源码，可以对照着看，效果更好。

此时，在翻开这第 2 卷时，您手边应该已经准备好了一个能正常调试 Dalvik 的环境。现在，就进入本卷，边学习边实践，揭开 Dalvik 内部的神秘面纱。

全书共分为 6 章：

第 1 章介绍类加载机制，包括其整体的工作流程和机制，详细讲解其中的三个阶段，并以一个实例验证了源码分析的结果；

第 2 章介绍 Dalvik 虚拟机中至关重要的内存管理机制，详细讲解其实现的两种算法；

第 3 章分析 JNI 模块的实现原理，在分析源码的基础上，细致入微地介绍为何用 JNI 编程会提升程序的执行效率；

第 4 章以反射机制的一个代码示例开始，介绍其涉及的 API，并从宏观（实现的三个模块）到微观（具体实现细节）详细介绍了反射机制；

第 5 章介绍实现解释器的两种不同的技术，比较 Fast 解释器和 Portable 解释器的不同及各自的优势和劣势；

第 6 章从介绍最近在解释器中非常火的 JIT（即时编译）开始，到 JIT 的所谓的前端分析，再到 JIT 的后端代码生成，为本书画上一个圆满的句号。

本书主要由哈尔滨工程大学吴艳霞、张国印负责编写，参与本书编写和校核工作的还有汪永峰、王彦璋、谢东良、于成、张婷婷、苗施亮、许圣明、檀凯，这里对他们的辛勤工作表示衷心的感谢。

本书主要是针对高级 Android 应用开发工程师、Android 系统开发工程师、Android 移植工程师及对 Android Dalvik 虚拟机源码实现感兴趣的读者参考使用。

编　者

2014 年 5 月

目录

第 1 章

类加载模块的原理及实现

本章内容提要

✍ 什么是类加载机制？

✍ 类加载机制具有怎样的功能？

✍ 类加载的输入输出是什么？

✍ 类加载机制的工作流程是什么？

✍ 类加载机制的函数具体实现是怎样的？

✍ 类加载机制是如何与解释器进行交互的？

本章主要围绕以下 6 点问题展开讨论，1.1 节主要回答了什么是类加载、类加载的具体功能以及机制的输入与输出；1.2 节从一个整体宏观的角度介绍了类加载机制的整体工作流程，给出了类加载机制的工作阶段划分；在接下来的三节中，分别对类加载机制的三个关键阶段进行详细的分析阐述，详细讲述了三个工作阶段的功能原理以及相关的函数源码分析；最后一节将会通过一个虚拟机运行实例，介绍类加载机制的产物是如何被解释器引用并执行，并从这一角度展示类加载机制与其他功能模块交互的方式。

1.1 类加载机制概述

谈起类加载，想必大多数读者比较陌生，类加载机制究竟具有怎样的功能以及它在虚拟机中是如何工作的是本章具体讨论的内容。

我们都知道程序是由机器指令和数据组成的，对于一个真实的机器，通常由人工将指令和数据交付给机器，再由机器按照指令去对数据进行计算。但是对于虚拟机来说，在其模拟真实机器执行程序之前，需要一种加载机制将程序的指令和数据装载进入虚拟机内部的运行时环境，使虚拟机中的执行模块可以根据程序执行的需要随时取得目标指令和相关数据，以完成程序的执行任务。对于 Java 虚拟机而言，这一种将类数据装载进入虚拟机运行时环境的过程就称为类加载。具体到 Dalvik 虚拟机，类加载机制的主要功能就是将应用程序中 Dalvik 操作码以及程序数据提取并加载到虚拟机内部，以保证程序的正确运行。

类加载机制在应用程序和执行模块之间建立起了一个桥梁，对于整个 Dalvik 虚拟机架构来说，类加载机制处在一个承上启下的位置。图 1.1 反映了类加载机制在 Dalvik 虚拟机执行过程中所承担的重要作用。

类加载机制作为 Dalvik 虚拟机一个重要的功能模块，其输入输出是什么呢？在第 1 卷第 3 章中，已经对 Dex 文件进行了介绍，在此就不再赘述。简单来说，Dex 文件就是 Android

应用程序的可执行文件,里面封装了 Dalvik 字节码以及程序数据,而类加载机制就是负责提取装载 Dex 文件中的指令与数据。因此,Dex 文件是 Dalvik 虚拟机的输入文件,更是类加载机制的主要处理对象。既然 Dex 文件是类加载机制的输入文件,那么类加载机制的输出又是什么呢?事实上,类加载机制的输出是一个名为 ClassObject 的数据结构实例对象,类加载将目标类的各项资源数据与 ClassObject 数据结构下的各个成员变量以指针的形式进行关联,执行模块只需通过该数据结构即可获得目标类所有的运行时数据,使程序的顺利执行成为可能。

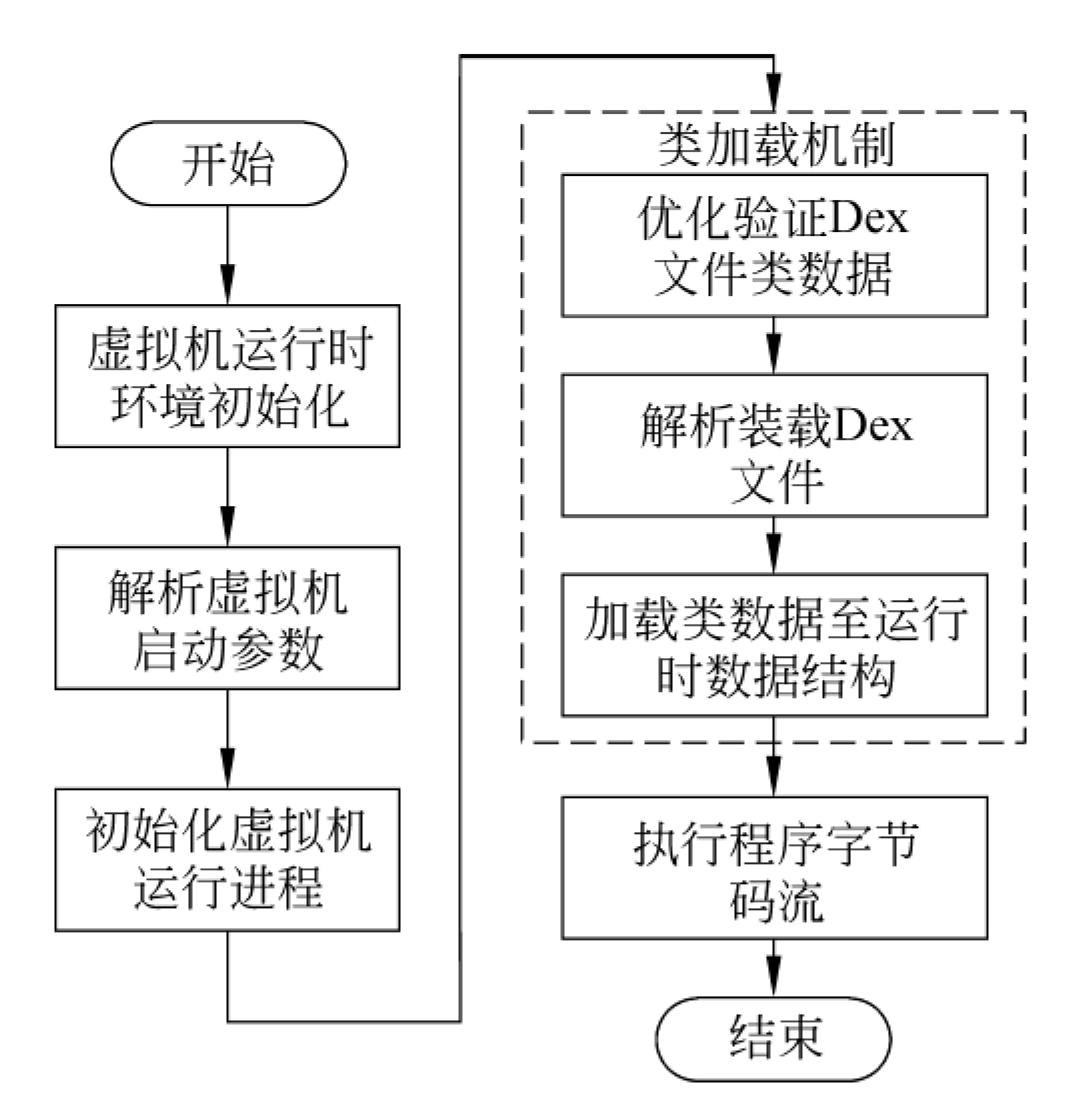


图 1.1　Dalvik 虚拟机程序执行流程图

点拨　程序是由指令序列组成的,告诉计算机如何完成一个具体的任务。程序是软件开发人员根据用户需求开发的、用程序设计语言描述的适合计算机执行的指令(语句)序列。而对于 Dalvik 虚拟机来说,它所能"读懂"的指令是 Dalvik 操作码,其介绍可以参见第 1 卷第 3 章。

1.2　类加载机制整体工作流程介绍

在了解了类加载机制的主要功能以及输入输出产物后,我们不禁要问,类加载机制在工作过程中主要有哪几项关键工作以及其工作流程又是什么?经过对类加载机制源码的分析研究,类加载机制工作的主要内容以及其整体的工作流程主要分为以下三点。

(1) 对 Dex 文件进行验证并优化,验证的目的是对 Dex 文件中的类数据进行安全性、合法性检验,为虚拟机的安全稳定运行提供保证;而优化的目的则是根据当前设备平台特性对程序中的字节码进行优化替换并为 Dex 文件增加辅助信息,最后输出经过优化的 Odex 文件,使之可以代替原有的 Dex 文件更加高效地被虚拟机执行。

(2) 对优化后的 Odex 文件进行解析,其目标就是通过在内存中创建专用的数据结构描述表示该 Odex 文件,使虚拟机对目标 Odex 文件中各个部分的类数据都是可达的,为随后实际加载某一指定类做好数据准备。

(3) 对指定类进行实际加载,其功能是实时根据 Dalvik 虚拟机执行需要从已被解析的 Dex 文件中提取二进制 Dalvik 字节码并将其封装进运行时数据结构,以供解释器解释执

行。而该运行时数据结构实际上是一个 ClassObject 结构体对象，也称为类对象，该数据结构用于封装程序类的所有运行时数据信息。当虚拟机执行一个类方法时，解释器将引用并执行类对象中封装的方法操作码，进而达到完成程序要求的执行目标。由此可见，类对象实例在程序运行的过程中承担着不可替代的重要作用。

图 1.2 简要描述了类加载机制的整体工作流程。

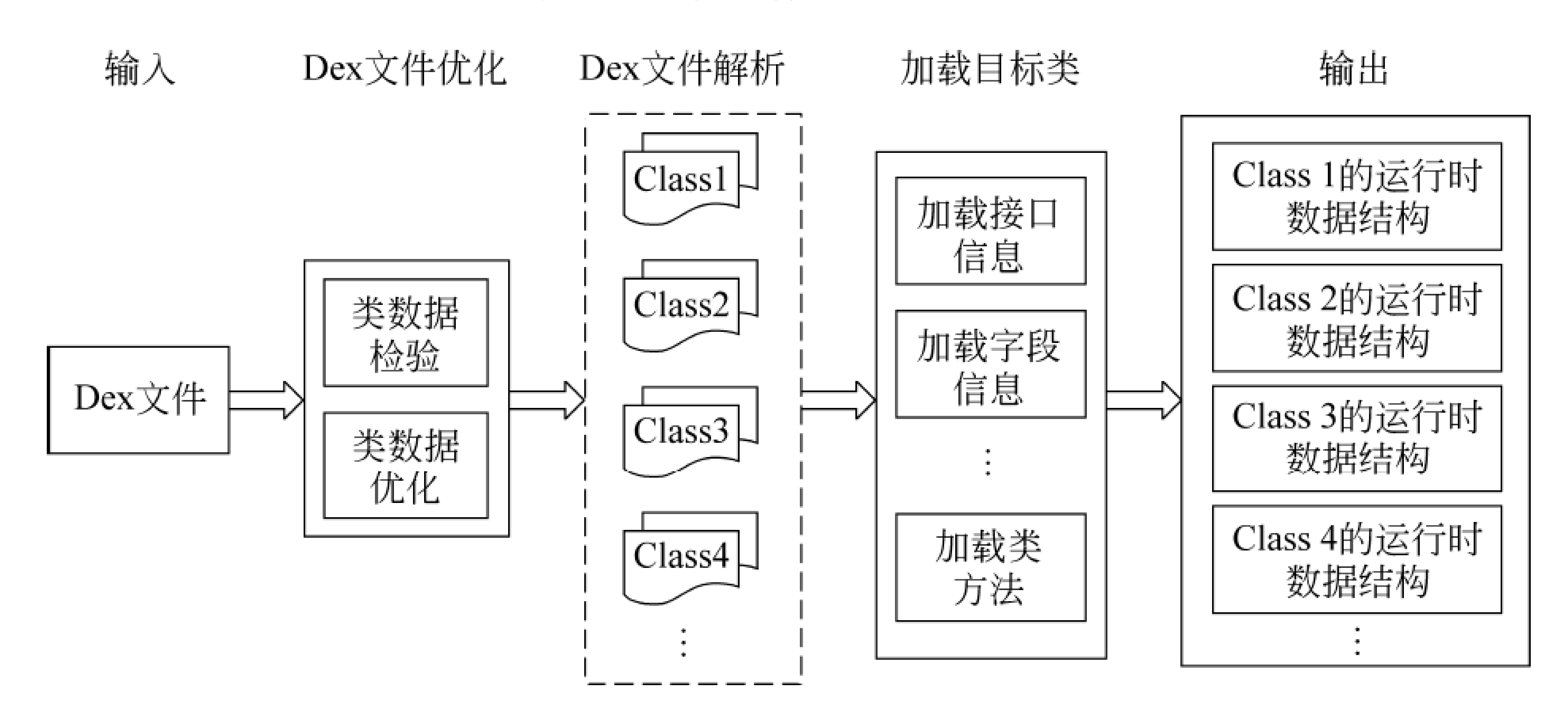


图 1.2　类加载机制整体工作概况图

在随后的内容中将分别从 Dex 文件的优化与验证、Dex 文件解析以及类的实际加载等三方面(关键内容)展开介绍，以帮助读者对类加载机制的整体功能、设计原理以及函数工作流程有一个较为深入的理解。

1.3　Dex 文件的优化与验证

事实上，Dex 文件中类数据的优化与验证是 Dex 文件解析工作的一部分，但由于该优化与验证工作在 Dalvik 虚拟机中被前置，使之成为 Dalvik 区别于其他 Java 虚拟机的重要特征，因此这部分工作具有较高的重要性，其工作效果的好坏在一定程度上决定了 Dalvik 虚拟机是否可以高效安全地运行，因此本书特将这部分内容抽取出来单独介绍。同时，Dalvik 虚拟机的优化技术一直是业界关注的焦点，如何进一步提高 Dalvik 虚拟机在嵌入式设备上的性能表现应该是未来工程开发人员工作的重点。

在这一节中，主要讲解了 Dex 文件的优化与验证技术的设计原理、关键数据结构以及其函数执行流程。希望通过对这三方面内容的介绍，可以让读者由表及里地了解该技术的功能原理以及具体实现。

1.3.1　Dex 文件优化验证的原理与实现

依据 Google 提供的开发文档的描述，在通常情况下，优化和验证 Dex 文件最安全且最便捷的方式就是在虚拟机中直接加载目标 Dex 文件并运行其中包含的所有类，因为一旦加载失败或是运行失败，就意味着 Dex 文件优化验证的失败。因此，虚拟机究竟是使用何种方法，以较为高效的手段实现了对 Dex 文件的验证与优化，这将是一个非常值得深入研究的问题。为了得到最为直观、真实的结论，需要从 Android 源码入手，以彻底弄清问题的本质。

根据对优化机制的源码研究发现：Dalvik 虚拟机的优化验证工作独立于程序的执行，同时优化验证这两部分功能也被整合成为一个功能模块并且类加载机制在加载工作的初期通过类似接口调用的方式调用这个优化模块对目标 Dex 文件进行优化。为了解决资源占用问题，Android 系统将会新建一个虚拟机用于实现相应的功能，在 Dex 文件的优化验证工作结束并正常输出优化文件后，系统将释放该虚拟机占用的所有资源。

同时，为了更大程度地保证原 Dex 文件的数据的安全以及优化机制的独立性，优化机制并不直接改写原 Dex 文件，而是重新创建一个后缀为.Odex 的空文件并以严格的格式要求将所有的优化信息写入该文件，主要包括依赖库关系、寄存器映射关系以及类的索引关系，这些关系的建立会大大提高类加载机制的执行效率，同时，在优化过程中还会根据平台特性对原 Dex 文件中部分字节码进行替换（例如，对字段的访问方式由查找改为直接引用）以提高程序执行速率，最后再将重写的 Dex 文件也写入 Odex 文件（1.3.2 节将详细介绍 Odex 文件结构以及各部分功能）。Odex 文件作为优化机制的输出将会取代原 Dex 文件并作为直接的可执行文件被其他功能模块（例如，内存管理模块、解释器模块等）调用。Dex 文件的优化机制大致的工作流程如图 1.3 所示。

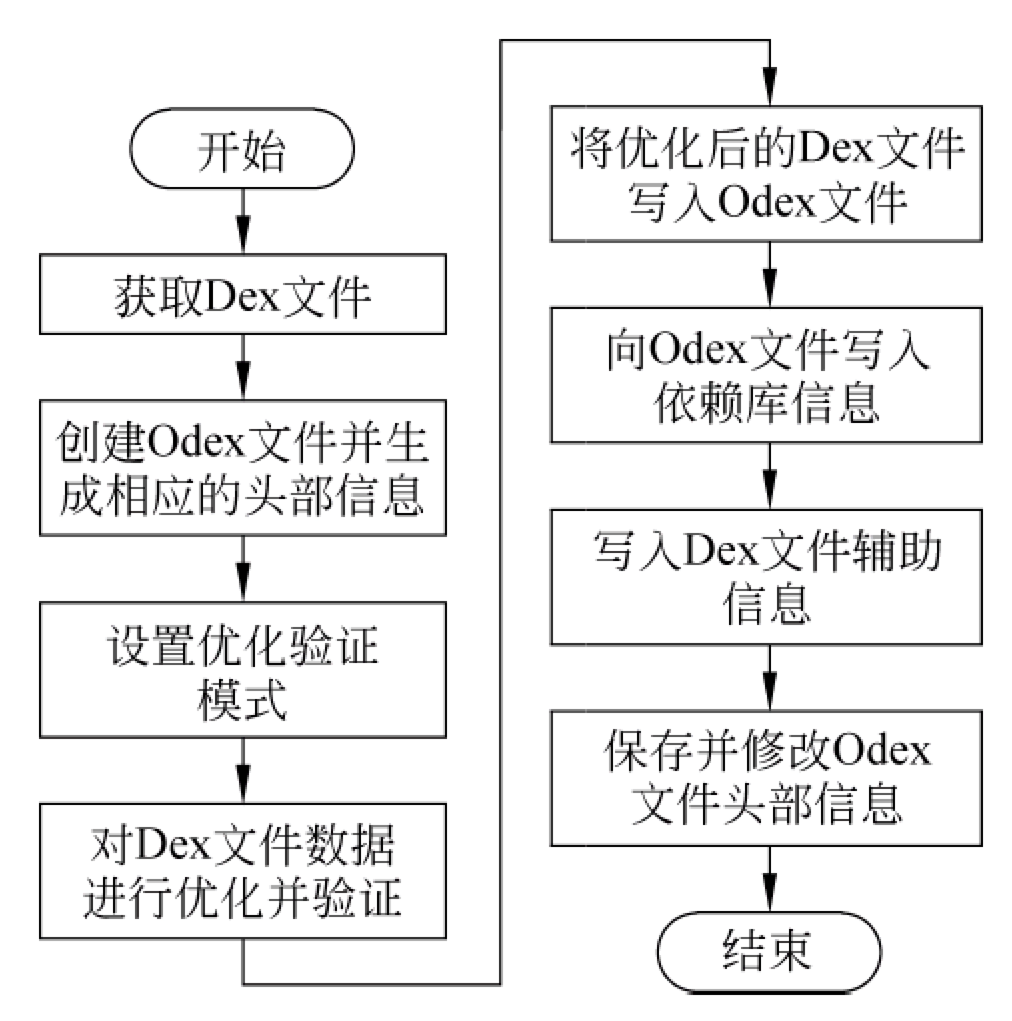


图 1.3　优化机制工作流程图

Dex 文件的优化与验证机制在 Dalvik 虚拟机中被设计成一个独立的系统工具，这样做的好处是使该机制具有比较高的独立性，使整个机制模块性更好，在一定程度上降低了整个 Android 系统的冗余。同时，也提供了一个技术参考，在适应低性能的移动平台时，应该尽量采用这种优化手段以提高效率。

点拨　Android 设备在第一次启动程序时往往耗费较长的时间，实际上，在这期间虚拟机对目标 Dex 文件进行了验证与优化，并为之生成了相应的 Odex 文件。当用户再次启动应用程序时，新生成的 Odex 文件将会代替原有的 Dex 的文件被虚拟机引用执行，由于不需要再次对程序数据进行验证优化，极大地缩短了启动时间。

1.3.2　Odex 文件结构分析

直观上 Odex 文件在 Dex 文件的原有结构上进行了扩充，即在 Dex 文件前拼接了 Odex 文件头部信息，还在 Dex 文件尾部拼接了依赖库、寄存器映射关系以及类的哈希索引等辅助信息。其结构对比如图 1.4 所示。

通过 Odex 文件的头部信息可以更好地了解一下 Odex 的文件结构以及各部分数据含义，表 1.1 为 Odex 文件头 DexOptHeader 在 Dexfile.h 文件中的定义。

在表 1.1 中，DexOptHeader 结构中的 magic 字段与 DexHeader 结构中的 magic 字段类似，都是用于标识文件；dexOffset 字段表示原 Dex 文件起始位置的偏移量，实际上它就等于 DexOptHeader 结构体的大小 0x28；dexLength 字段表示 Dex 文件的总长度，通过这两个字段可以非常快速定位并读取 Dex 文件；depsOffset 字段表示依赖库起始的偏移量；

depsLength 表示依赖库的总长度；optOffset 字段表示优化数据的起始偏移量；optLength 字段表示优化信息的总长度，而对于类加载机制非常关键的类索引信息就封装在这部分优化信息中；flags 字段为一个标识，其用于标示 Dalvik 虚拟机加载 Odex 文件时优化与验证选项；checksum 字段为 Odex 文件的校验和。

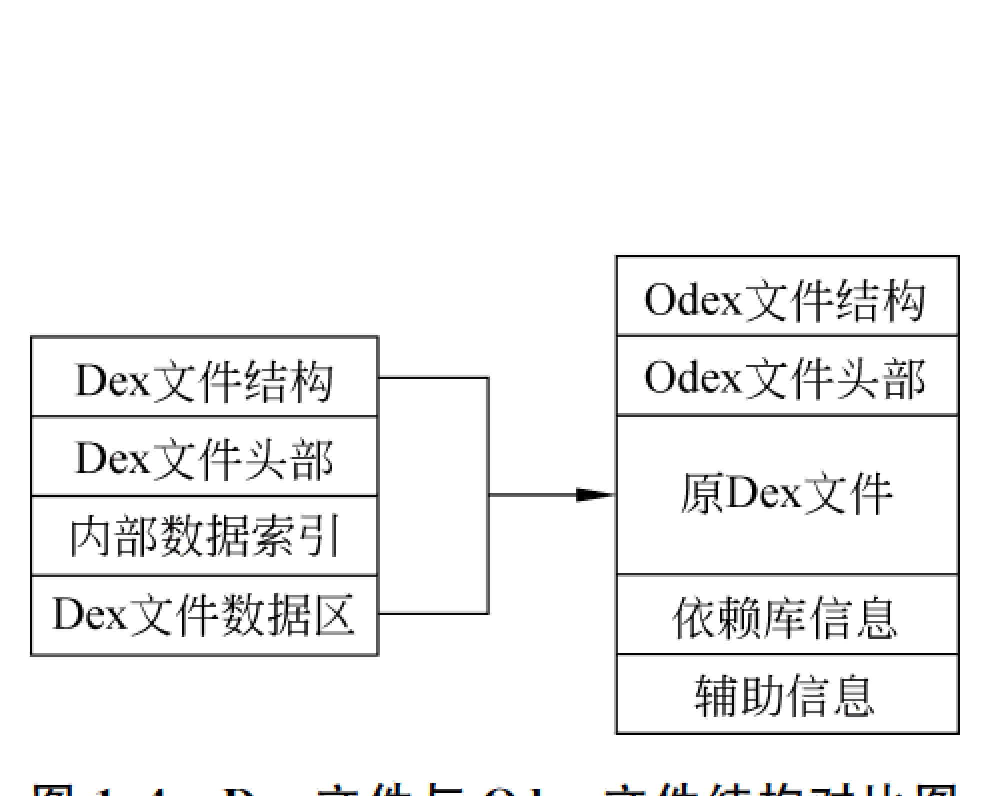


图 1.4　Dex 文件与 Odex 文件结构对比图

表 1.1　DexOptHeader 数据结构定义

变量类型	变量名称	描　述
u1	magic[8]	Odex 文件版本标识
u4	dexOffset	Dex 文件头偏移量
u4	dexLength	Dex 文件总长度
u4	depsOffset	Odex 文件依赖库列表偏移量
u4	depsLength	依赖库信息总长度
u4	optOffset	优化数据信息偏移量
u4	optLength	优化数据总长度
u4	flags	标识位
u4	checksum	文件校验和

通过这个头部信息，虚拟机可以非常高效地查找 Dex 文件中的各类信息，极大提高了执行效率，另外，Dalvik 虚拟机对 Dex 文件所进行的优化工作主要体现在依赖库和辅助信息两部分上，因此，下文将会对这两部分内容的功能进行介绍。

1. 依赖库信息

依赖库顾名思义，就是指该 Dex 文件所需要链接的本地函数库，Dalvik 虚拟机在程序执行前期通过优化机制将这部分整合到 Odex 文件中，可以在一定程度上提高程序的执行效率，表 1.2 为依赖库 Dependence 结构体定义。

表 1.2　Dependence 数据结构定义

变量类型	变量名称	描　述
u4	modWhen	时间戳
u4	crc	校验信息
u4	DALVIK_VM_BUILD	虚拟机版本号
u4	numDeps	依赖库个数
u4	len	Name 长度
u4	name[len]	依赖库名称
KSHA1DigestLen	signature	SHA-1 值

在表 1.2 中，modWhen 用来记录 Dex 文件优化前的时间戳，crc 为 Dex 文件优化前的 crc 校验值，DALVIK_VM_BUILD 值表示的是虚拟机版本号；不同版本的 Android 系统定义不同，例如，Android 2.2.3 为 19、Android 2.3 为 23 以及 Android 4.0.4 则为 27。numDeps 字段所代表的含义为该 Dex 文件的依赖库个数，其中 table 结构体的个数正是由

numDeps 决定的，也可以理解为每个依赖库都对应一个 table 结构体对象，在该结构体中，len 表示依赖库名称的长度、name 为依赖库名以及 signature 表示 SHA-1 签名。

2. 类索引信息

类索引信息的建立是优化机制的重要工作之一，在该索引表中，优化机制为 Dex 文件中的每一个类配置了一个 table 结构体对象，在这个对象中记录了类描述符哈希值、类描述符在 Dex 文件中偏移地址以及类定义区的偏移地址，类加载机制通过这些信息可以非常快速地定位类资源地址并加载类。同时，通过哈希查找的方式极大地提高了类加载机制的查找效率，表 1.3 为 DexClassLookup 结构体定义。

表 1.3 DexClassLookup 数据结构定义

变量类型	变量名称	描　述
int	size	表大小
int	numEntries	表项入口数量
u4	classDescriptorHash	类描述符的哈希值
u4	classDescriptorOffset	Dex 文件中该类描述符的偏移位置
u4	classDefOffset	Dex 文件中该类定义偏移位置

在表 1.3 中，numEntries 是一个比较特别的数目，它虽然表示的是表的项数，但实际上这个数值是通过 dexRoundUpPower2()函数生成。

点拨 dexRoundUpPower2()函数是源自斯坦福大学的一个算法——用于求比一个数大的最小的 2 的整数次幂，例如：当数为 6 时，该算法计算得到 8。这样做的结果会比 Dex 文件中类的数量大，但好处是降低了哈希冲突率。

1.3.3 函数执行流程

Dex 文件的优化始于 Android 源码中 frameworks 层的 PackageManagerService 类，该类实际上是通过 Installer 类实现对 apk 文件的安装、优化以及卸载等工作。而 Installer 类通过与 c 层的 installd 建立 socket 连接，使得在上层的 install、remove、dexopt 等功能最终由 installd 在底层实现。在 installd 中，do_dexopt 函数负责完成对 Dex 文件的优化，而 do_dexopt 函数将会调用 dexopt 函数去完成实际的优化工作。在 dexopt 函数中，首先完成一些必要的准备工作，比如声明关键变量，分析文件路径，而最关键的是创建了一个空文件并以 Odex 后缀结尾，由此，可以认定 dexopt 函数用于产生 Odex 文件，在完成准备工作后，将会委托 run_dexopt 函数完成实际的优化工作。

在 run_dexopt 函数中，可以看到这样一行代码：

```
execl(DEX_OPT_BIN,DEX_OPT_BIN,"--zip",zip_num,odex_num,input_file_name,dexopt_flags,(char*)NULL);
```

run_dexopt 函数通过使用 Execl 函数将优化工作委托给了由第一参数 DEX_OPT_BIN 宏定义所指出的可运行程序，DEX_OPT_BIN 宏定义的内容是：

```
static const char* DEX_OPT_BIN="/system/bin/dexopt";
```

而/system/bin/dexopt 中的 dexopt 程序的源码位于 Android 系统源码的 dalvik\dexopt 目录下，从目录结构上即可反映出 dexopt 优化程序是 Dalvik 虚拟机下第一级子程序，也正说明优化机制在 Dalvik 虚拟机中确实为一个相对独立的功能模块。

在清楚了优化机制的源头后，dexopt 优化程序的工作流程成为我们比较关注的一个问题，因此，本文将在此着重分析优化机制的具体实现过程并介绍优化机制中关键的技术点以及有代表性的实现细节。

dexopt 的主程序代码位于 dalvik\dexopt\OptMain. cpp 文件中，其中 extractAndProcessZip() 函数用于处理并优化 apk/jar/zip 文件中的 classes. dex，因此该函数将作为优化机制的主控函数，extractAndProcessZip()函数的实现代码如下。

代码清单 1.1 dalvik\dexopt\OptMain. cpp：extractAndProcessZip()函数源代码

```
static int extractAndProcessZip( int zipFd,int cacheFd,const char*  debugFileName,bool isBootstrap,const
char*  bootClassPath,const char*  dexoptFlagStr)
{
    /* 函数在执行初期声明相关的中间变量 */
    ZipArchive zippy;                   //用于描述 ZIP 压缩文件的数据结构
    ZipEntry zipEntry;                  //用于表示一个 ZIP 入口
    …
    off_t dexOffset;                    //用于表示在 Odex 文件中，原 Dex 文件的起始地址
    int err;                            //标示符
    int result=-1;                      //函数返回值
    int dexoptFlags=0;                  //优化标示符
    /* 设置默认的优化模式 */
    DexClassVerifyMode verifyMode=VERIFY_MODE_ALL;
    DexOptimizerMode dexOptMode=OPTIMIZE_MODE_VERIFIED;
    memset(&zippy,0,sizeof(zippy));     //对 zippy 对象进行置 0 操作
    /* 对入口参数 cacheFd 文件描述符所代表的输入文件进行为空判断，该文件必须保证为空，因为
       在后期要将优化后的数据写入该文件中 */
    if (lseek(cacheFd,0,SEEK_END) != 0) {
        LOGE("DexOptZ: new cache file '%s' is not empty",debugFileName);
        goto bail;
    }
   /* 当 cacheFd 所指文件为空，那么为其创建一个 Odex 文件的头部 */
    err=dexOptCreateEmptyHeader(cacheFd);
    if (err != 0)                       //对函数执行结果进行判断，如果失败则将返回
        goto bail;
  /* 取得 Odex 文件中原 Dex 文件的起始位置，实际就是一个 Odex 文件头部的长度，并将结果赋值给
     变量 dexOffset */
    dexOffset=lseek(cacheFd,0,SEEK_CUR);
    if (dexOffset<0)
        goto bail;
    /* 打开 ZIP 对象，在其中查找目标 Dex 文件 */
    if (dexZipPrepArchive(zipFd,debugFileName,&zippy) != 0) {
```

```
        LOGW("DexOptZ: unable to open zip archive '%s'",debugFileName);
        goto bail;
    }
/* 获取目标 Dex 文件的解压入口 */
    zipEntry=dexZipFindEntry(&zippy,kClassesDex);
    if (zipEntry==NULL) {
        LOGW("DexOptZ: zip archive '%s' does not include %s",
            debugFileName,kClassesDex);
        goto bail;
    }
    /* 获取相关 ZIP 入口信息 */
    if (dexZipGetEntryInfo(&zippy,zipEntry,NULL,&uncompLen,NULL,
                          NULL,&modWhen,&crc32) != 0)
    {
        LOGW("DexOptZ: zip archive GetEntryInfo failed on %s",
                debugFileName);
        goto bail;
    }
    ⋮
    /* 从 ZIP 文件将目标 Dex 文件解压出来,并写入 cacheFd 所指文件,此时 cacheFd 所指文件非空,包
       括一个 Odex 文件头部加上一个原始的 Dex 文件 */
        if (dexZipExtractEntryToFile(&zippy,zipEntry,cacheFd) != 0) {
        LOGW("DexOptZ: extraction of %s from %s failed",
            kClassesDex,debugFileName);
        goto bail;
    }
/*  根据入口参数 dexoptFlagStr,对验证优化需求进行分析,dexoptFlagStr
      实际上是一个字符串,记录了验证优化的要求 */
    if (dexoptFlagStr[0]!='\0') {
        const char*  opc;
        const char*  val;
            /* 设置验证模式 */
        opc=strstr(dexoptFlagStr,"v=");      /*  verification */
        if (opc !=NULL) {
         switch (*(opc+2)) {
         case 'n': verifyMode=VERIFY_MODE_NONE;          break;
         case 'r': verifyMode=VERIFY_MODE_REMOTE;        break;
         case 'a': verifyMode=VERIFY_MODE_ALL;           break;
         default:                                        break;
            }
        }
        /* 设置优化模式 */
        opc=strstr(dexoptFlagStr,"o=");      /*  optimization */
        if (opc !=NULL) {
```

```
            switch (* (opc+2)) {
            case 'n': dexOptMode=OPTIMIZE_MODE_NONE;        break;
            case 'v': dexOptMode=OPTIMIZE_MODE_VERIFIED;    break;
            case 'a': dexOptMode=OPTIMIZE_MODE_ALL;         break;
            case 'f': dexOptMode=OPTIMIZE_MODE_FULL;        break;
            default:                                        break;
                }
            }
    ⋮
    }
    /* 当完成了原 Dex 文件的提取以及验证优化选项的设置,即可以开始真正的优化工作,需要初始化
       一个虚拟机专门用于验证优化工作 * /
       if (dvmPrepForDexOpt(bootClassPath,dexOptMode,verifyMode,
               dexoptFlags) != 0)
       {
           LOGE("DexOptZ: VM init failed");
           goto bail;
       }
    /* 调用 dvmContinueOptimization 函数完成对 Dex 文件的验证与优化工作 * /
       if (!dvmContinueOptimization(cacheFd,dexOffset,uncompLen,
                   debugFileName,modWhen,crc32,isBootstrap))
       {
           LOGE("Optimization failed");
           goto bail;
       }
       result=0;                                   //设置返回值,0表示成功
       ⋮
       return result;                              //函数返回 }
```

从上面的源码中可以看到,extractAndProcessZip()函数首先会调用 dexOptCreateEmptyHeader()函数为 Odex 文件创建一个文件头用于描述 Odex 文件内容,随后主函数将调用 dexZipFindEntry()函数检查目标文件中是否拥有 classes. dex 文件,如果目标 Dex 文件存在,则通过 dexZipGetEntryInfo()函数读取 Dex 文件相关的验证信息,接着调用 dexZipExtractEntryToFile()函数提取 classes. dex 文件并写入 Odex 文件,在写入完毕后,主程序将会根据入口参数 dexoptFlagStr 解析检验与优化模式并将优化选项 dexOptMode 与验证 verifyMode 写入到全局变量中。至此,优化机制的准备工作基本结束。

在准备工作完成后,主函数调用 dvmPrepForDexOpt()启动并初始化一个虚拟机进程,而以后的所有优化操作都会在这个进程中完成,当 Odex 文件正常输出后,这个进程的所有资源都会被释放。当优化的进程准备完毕后,主函数将调用 dvmContinueOptimization()函数开始真正的验证与优化工作。

点拨 启用一个专门的进程用于负责 Dex 文件的优化与验证,这样做的好处是在一定程度上保证了虚拟机的安全运行,因为一个未经安全验证的程序,不能保证对其他虚拟机进程绝对安全。例如,恶意篡改共享数据、造成内存溢出等。

代码清单 1.2 dalvik\vm\analysis\DexPrepare.cpp :dvmContinueOptimization()函数源代码

```
bool dvmContinueOptimization(int fd,off_t dexOffset,long dexLength,
            const char* fileName,u4 modWhen,u4 crc,bool isBootstrap)
{
    /* 声明相关中间变量 */
    DexClassLookup* pClassLookup=NULL;
    RegisterMapBuilder* pRegMapBuilder=NULL;
    assert(gDvm.optimizing);
    LOGV("Continuing optimization (%s,isb=%d)",fileName,isBootstrap);
    assert(dexOffset >=0);                    //判断输入文件长度非 0
    /* 对目标文件进行合法性检验 */
    if (dexLength< (int) sizeof(DexHeader)) {
    /* 一个 Dex 文件的长度不能小于其文件头的长度 */
        LOGE("too small to be DEX");
        return false;
    }
    /* Odex 文件中的 Dex 文件的起始偏移量不能小于 Odex 文件头的长度 */
    if (dexOffset< (int) sizeof(DexOptHeader)) {
        LOGE("not enough room for opt header");
        return false;
    }
    ⋮
        /* 将 fd所指文件映射到某一位置,该位置的起始地址为 mapAddr,其大小就为 fd所指文件大
小,即一个 Odex 文件头部加上一个 Dex 文件长度 */
        mapAddr=mmap(NULL,dexOffset+dexLength,
            PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
        if (mapAddr==MAP_FAILED) {
            LOGE("unable to mmap DEX cache: %s",strerror(errno));
            goto bail;
        }
        /* 设置相关的优化验证选项 */
        bool doVerify,doOpt;
        if (gDvm.classVerifyMode==VERIFY_MODE_NONE) {
            doVerify=false;
        } else if (gDvm.classVerifyMode==VERIFY_MODE_REMOTE) {
            doVerify=!gDvm.optimizingBootstrapClass;
        } else /* if (gDvm.classVerifyMode==VERIFY_MODE_ALL) */ {
            doVerify=true;
        }
        if (gDvm.dexOptMode==OPTIMIZE_MODE_NONE) {
            doOpt=false;
        } else if (gDvm.dexOptMode==OPTIMIZE_MODE_VERIFIED ||
```

```
                gDvm.dexOptMode==OPTIMIZE_MODE_FULL) {
            doOpt=doVerify;
        } else /* if (gDvm.dexOptMode==OPTIMIZE_MODE_ALL) */ {
            doOpt=true;
        }
    /* 调用 rewriteDex 函数对目标文件进行优化验证,其主要内容包括:字符顺序调整、字节码
     替换、字节码验证以及文件结构重新对齐 */
        success=rewriteDex(((u1*) mapAddr)+dexOffset,dexLength,
                        doVerify,doOpt,&pClassLookup,NULL);
     …    /* 对当前文件进行 8 字节对齐,并准备写入 */
    off_t depsOffset,optOffset,endOffset,adjOffset;
    int depsLength,optLength;
    u4 optChecksum;
    depsOffset=lseek(fd,0,SEEK_END);                //取得 fd所指文件的总长
    if (depsOffset<0) {
        LOGE("lseek to EOF failed: %s",strerror(errno));
        goto bail;
    }
    /* 根据 fd所指文件总长使 depsOffset(dependency list 的起始地址)为 8 的倍数 (adjOffset 应该大
于等于 depsOffset) */
    adjOffset=(depsOffset+7) & ~(0x07);
    if (adjOffset !=depsOffset) {
        LOGV("Adjusting deps start from %d to %d",
            (int) depsOffset, (int) adjOffset);
        depsOffset=adjOffset;
        lseek(fd,depsOffset,SEEK_SET);
    }
    /* 写入依赖库信息 */
    if (writeDependencies(fd,modWhen,crc) !=0) {
        LOGW("Failed writing dependencies");
        goto bail;
    }
 ⋮
    /* 写入其他优化信息,包括类索引信息以及寄存器映射关系 */
    if (!writeOptData(fd,pClassLookup,pRegMapBuilder)) {
        LOGW("Failed writing opt data");
        goto bail;
    }
 ⋮
    /* 对 Odex 文件的头部内容进行修正 */
    DexOptHeader optHdr;
    memset(&optHdr,0xff,sizeof(optHdr));
    memcpy(optHdr.magic,DEX_OPT_MAGIC,4);
    memcpy(optHdr.magic+4,DEX_OPT_MAGIC_VERS,4);
```

```
        optHdr.dexOffset= (u4) dexOffset;
        optHdr.dexLength= (u4) dexLength;
        optHdr.depsOffset= (u4) depsOffset;
        optHdr.depsLength= (u4) depsLength;
        optHdr.optOffset= (u4) optOffset;
        optHdr.optLength= (u4) optLength;
        ⋮
        return result;
    }
```

dvmContinueOptimization()函数首先对目标文件进行格式检验,并调用 mmap()函数将原 Dex 文件整体映射至内存,再调用 rewriteDex()函数并根据全局变量中的 dexOptMode 与 classVerifyMode 字段所规定的优化要求来重写映射文件,这里的重写内容包括字符顺序调整、字节码替换、字节码验证以及文件结构重新对齐等工作。当 rewriteDex()函数正常返回后,dvmContinueOptimization()函数将会依次调用 writeDependencies()函数建立依赖库和 writeOptData()函数写入辅助数据。当以上几个步骤结束后,Odex 文件的各个关键内容都已经具备,程序的最后将根据现实情况写入 Odex 文件头部信息并保存。

上面两个函数是位于机制函数调用链的上层,通过对源码的展示分析,相信读者已经对机制的关键的执行流程有了一定的认识。由于篇幅所限,再加上优化机制源码量较为巨大,本书在此就不再对源码进行更加深入的介绍展示了,希望读者能够结合下载的源码对文中出现的各个关键的功能点函数进行学习,将会非常有助于理解机制中一些关键技术点的具体实现细节。

点拨 对于一个 Dex 文件来说,如果在 cache 中存在一个与之对应的 Odex 文件,虚拟机在接收到运行指令时,会直接引用执行相对应的 Odex 文件,这样就避免了二次验证与优化,减少了程序启动时间。

1.4 Dex 文件的解析

Dalvik 虚拟机与标准 Java 虚拟机最为直观的不同在于其输入文件格式,Dalvik 虚拟机采用由多个 Class 文件整合而成的 Dex 文件,其结构设计与意义已在第 1 卷第 3 章中进行了介绍,此处不再赘述。由于多个类被整合到一个 Dex 文件中,且在 Dex 文件中类与类之间不但没有明确的界限,甚至还有共享数据的情况,因此,这就要求类加载机制在实际加载一个目标类之前要对 Dex 文件进行一系列预处理,使 Dex 文件对于虚拟机来说是可读的。简单来说,Dex 文件解析的主要目的是对 Dex 文件进行读取分析并通过建立一个 DexFile 结构体的实例对象专门用于描述该 Dex 文件,使实际的类加载函数可以通过该数据结构对目标类全部的数据进行索引并提取以完成类的实际加载工作。

1.4.1 DexFile 数据结构简析

为目标 Dex 文件生成一个 DexFile 数据结构实例对象是 Dex 文件解析工作的重要目标,在第 1 卷第 3 章中,已经对 Dex 文件的结构进行了详细的介绍,想必读者读到此处时仍

还能回忆起些许内容。下面在介绍解析工作的原理和实现之前，先将对 DexFile 数据结构进行一个简要的介绍，如果读者对该结构体中各个成员变量以及其数据结构感到陌生，建议重新回顾一下第 1 卷第 3 章的内容。

DexFile 结构体的定义如表 1.4 所示，从中可以发现该数据结构的成员变量所代表的内容以及其排列顺序与经过优化的 Dex 文件的结构非常相似，Dex 文件中的各个数据部分似乎都可以和 DexFile 结构体中的各个成员变量形成相对应关系。

表 1.4　struct DexFile 结构体成员变量

成员变量类型	变 量 名	意　　义
const DexOptHeader*	pOptHeader	优化数据头部
const DexHeader*	pHeader	Dex 文件头部
const DexStringId*	pStringIds	指向字符串索引区
const DexTypeId*	pTypeIds	指向类型索引区
const DexFieldId*	pFieldIds	指向字段索引区
const DexMethodId	pMethodIds	指向方法索引区
const DexProtoId*	pProtoIds	指向原型索引区
const DexClassDef*	pClassDefs	指向类定义区
const DexLink*	pLinkData	指向连接数据区
const DexClassLookup*	pClassLookup	指向类索引
const u1*	baseAddr	基地址

写到这里读者就应该知道，虚拟机正是通过将目标 Dex 文件与一个 DexFile 数据结构进行关联，使得在虚拟机内部，负责类的实际加载的功能函数可以通过该数据结构实现对 Dex 文件中各类数据的查找获取工作。也正是利用这种方式，虚拟机实现了对一个存储在内存中不可读的数据块——Dex 文件进行解析的目的。

1.4.2　Dex 文件解析流程概述

当虚拟机获得了程序中的 Classes.dex 文件后，将首先对这个 Dex 文件进行初步的验证工作，主要包括：验证 Magic 数、校验 SHA-1 签名、计算 Dex 校验和等几个方面，其目的主要是检验该 Dex 文件是否符合 Android 系统规范。当校验的工作完毕后，将对 Dex 文件进行优化和验证并输出优化的产物——Odex 文件，Dex 优化与验证在 1.3 节已经进行了详细的分析和介绍。在接下来的工作中，Odex 文件将取代原 Dex 文件被加载机制进行解析。

当优化的工作结束后，虚拟机将会开始对优化过的 Dex 文件进行解析，首先需要对优化后的 Odex 文件进行完整性检验，以确保该 Odex 文件是完整并且合乎 Android 系统规范。随后，虚拟机将对目标 Odex 文件中的优化数据进行解析，其主要目的是将在对 Dex 文件优化期间生成的优化数据，例如：类索引信息、寄存器映射关系等，提前与 DexFile 数据结构中的个别成员变量进行关联。完成对优化数据的处理之后，虚拟机将对原 Dex 数据进行解析，其做法仍然是将 DexFile 数据结构中其他各个成员变量与 Dex 文件的各个数据部分相关联，使虚拟机能够更加高效地对 Dex 文件中的类数据进行查找并获取，其关联效果如图 1.5 所示。

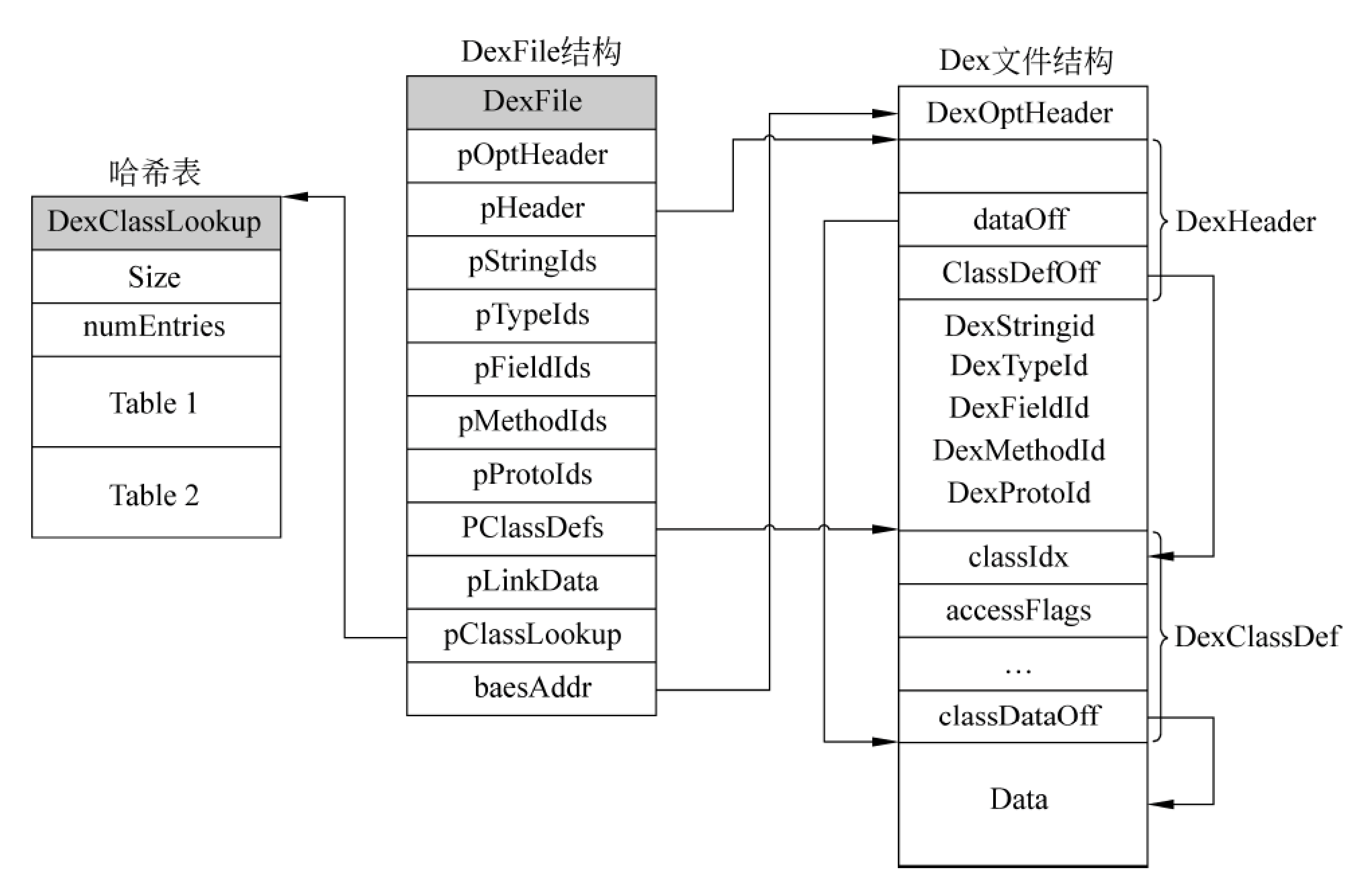


图 1.5　Dex 文件与 DexFile 数据结构映射关系图

当解析函数正确输出 DexFile 数据结构的一个实例对象之后，虚拟机将再次对 Dex 文件进行校验并计算 SHA-1 值，最后保存相关设定并将该实例对象返回。至此，Dex 文件的解析工作结束，其大致的工作流程如图 1.6 所示。

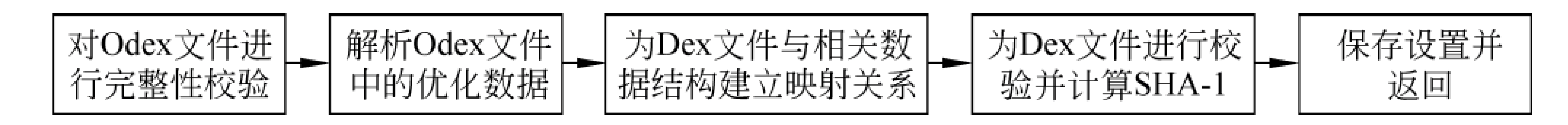


图 1.6　Dex 文件解析工作流程示意图

在 1.4.3 节中，将从源码实现的角度对 Dex 文件解析这部分工作进行一个详细的分析，同时也希望读者能够结合这部分源码进行学习，以加深对这部分内容的理解。

点拨　DexFile 结构体是用于描述内存中的 Dex 文件，虚拟机可以通过该数据结构快速访问 Dex 文件中各部分的数据。然而，这一切的功能实现是需要建立在一个严格的格式体系之中。由此可见，系统架构的设计工作是非常重要的。

1.4.3　函数执行流程

Dex 文件解析的工作主要由位于虚拟机源码目录 dalvik/vm/RawDexFile.cpp 中的 dvmRawDexFileOpen 函数完成，在这部分工作中称 dvmRawDexFileOpen 函数为主控函数，因为在它的调控下，各个功能点函数共同配合完成了对 Dex 文件进行解析的工作。dvmRawDexFileOpen 函数源码如下。

代码清单 1.3　dalvik\vm\RawDexFile.cpp：dvmRawDexFileOpen()函数源代码

```
int dvmRawDexFileOpen(const char* fileName,const char* OdexOutputName,
                      RawDexFile** ppRawDexFile,bool isBootstrap)
{
    /*声明函数中间的执行变量*/
```

```
DvmDex* pDvmDex=NULL;                //用于在虚拟机中描述解析的 Dex 文件
char* cachedName=NULL;               //用于保存执行期间产生的优化 Dex 文件名
int result=-1;                       //设置函数返回值,0 表示成功
int dexFd=-1;                        //初始化目标 Dex 文件的文件描述符
int optFd=-1;                        //初始化优化 Dex 文件的文件描述符
u4 modTime=0;                        //初始化文件修改时间参数
u4 adler32=0;                        //初始化校验和变量
size_t fileSize=0;                   //标示文件大小
bool newFile=false;                  //标示虚拟机是否需要对 Dex 文件进行优化
bool locked=false;                   //用于标示优化进程占用
/* fileName 是 dvmRawDexFileOpen 函数最关键的入口参数,它是一个字符串,记录了目标
 Dex 文件在文件系统中的绝对路径,主函数通过 open 函数根据 fileName 参数将目标文件进行读
入至虚拟机中 */
dexFd=open(fileName,O_RDONLY);
if (dexFd<0) goto bail;
⋮
dvmSetCloseOnExec(dexFd);
/* 对 Dex 文件的合法性与正确性进行检验 */
if (verifyMagicAndGetAdler32(dexFd,&adler32)<0) {
    LOGE("Error with header for %s",fileName);
    goto bail;
}
/* 记录文件修改时间并赋值给 modTime 参数 */
if (getModTimeAndSize(dexFd,&modTime,&fileSize)<0) {
    LOGE("Error with stat for %s",fileName);
    goto bail;
}
/* 根据目标 Dex 文件名为其产生相应的优化文件名并赋值给 cachedName */
if (OdexOutputName==NULL) {
    cachedName=dexOptGenerateCacheFileName(fileName,NULL);
    if (cachedName==NULL)
        goto bail;
} else {
    cachedName=strdup(OdexOutputName);
}
LOGV("dvmRawDexFileOpen: Checking cache for %s (%s)",
        fileName,cachedName);
/* 尝试根据 cachedName 所指的优化文件名在 cache 中查找并读取优化文件,如果读取失败或
 是当前的优化文件有误,则将要重新对 Dex 文件进行优化 */
optFd=dvmOpenCachedDexFile(fileName,cachedName,modTime,
    adler32,isBootstrap,&newFile,/* createIfMissing= */true);
if (optFd<0) {
    LOGI("Unable to open or create cache for %s (%s)",
            fileName,cachedName);
    goto bail;
```

```
    }
    locked=true;/* 在前面提到,如果 Odex 文件打开失败,则虚拟机将把 newFile 参数置为真。在此处
虚拟机将根据 newFile 的值以决定是否需要对 Dex 文件进行优化 * /
    if (newFile) {
        u8 startWhen,copyWhen,endWhen;
        bool result;
        off_t dexOffset;
        dexOffset=lseek(optFd,0,SEEK_CUR);
        result=(dexOffset>0);
        if (result) {
            startWhen=dvmGetRelativeTimeUsec();
         /* 将 dexFd 所指文件复制至 optFd 所指文件中 * /
            result=copyFileToFile(optFd,dexFd,fileSize)==0;
            copyWhen=dvmGetRelativeTimeUsec();
        }
        /* 调用 dvmOptimizeDexFile 函数对 optFd 所指文件进行优化 * /
        if (result) {
            result=dvmOptimizeDexFile(optFd,dexOffset,fileSize,
                fileName,modTime,adler32,isBootstrap);
        }
   ⋮
    }
/* 当 Dex 文件的优化结束后,将会调用 dvmDexFileOpenFromFd 函数对该 Dex 文件进行解析 * /
    if (dvmDexFileOpenFromFd(optFd,&pDvmDex) != 0) {
        LOGI("Unable to map cached %s",fileName);
        goto bail;
    }
   ⋮
    LOGV("Successfully opened '%s'",fileName);
   ⋮
    /* 对入口参数 ppRawDexFile 进行设置,ppRawDexFile 变量是一个 RawDexFile * 类型的指针,其作用
     是用于保存当前处理的 Dex 文件的相关信息 * /
    * ppRawDexFile=(RawDexFile * ) calloc(1,sizeof(RawDexFile));
    /* 保存优化文件名 * /
    (* ppRawDexFile)->cacheFileName=cachedName;
    /* 保存 DvmDex 数据结构 * /
    (* ppRawDexFile)->pDvmDex=pDvmDex;
    cachedName=NULL;                      //释放 cachedName 变量
    result=0;                             //设置返回值,0表示成功
    return result;
}
```

dvmRawDexFileOpen 函数首先通过调用 verifyMagicAndGetAdler32 函数完成对 Dex 文件的 magic 字段以及校验信息的验证工作,如果该函数正确返回,证明该 Dex 文件为一个合法文件。

点拨 魔数(magic)是为了方便虚拟机识别目标文件是否为合格的 Dex 文件,在 Dex 文件中,magic 为 dex/n035/0;而在标准的 Java Class 文件中,其 magic 为 CAFEBABE。

随后主函数通过调用 dexOptGenerateCacheFileName 函数生成该 Dex 文件的优化文件名，紧接着调用 dvmOpenCachedDexFile 函数，根据这个 Dex 文件的优化文件名在 dalvik_cache 中查找是否存在该 Dex 文件的优化文件(虚拟机在第一次执行一个 Dex 文件时会对这个 Dex 文件进行优化并生成 Odex 文件，置放于 cache 中，当再次运行该 Dex 文件时，将跳过优化的步骤，直接找到 Odex 文件并运行之)，如果不存在，主函数在设置相关的优化参数后，将调用 dvmOptimizeDexFile 函数完成对 Dex 文件的优化工作并在 cache 中生成 Odex 文件。至此，主函数完成了对 Dex 文件的检验与优化工作。这部分的工作重点是对优化的 Dex 文件进行解析，那么该解析工作是由哪个函数具体承担呢?

dvmRawDexFileOpen 函数在完成对目标 Dex 文件的优化后，将会调用 dvmDexFileOpenFromFd 函数完成 Dex 文件的后续解析工作，其源码如下。

代码清单 1.4　dalvik\vm\DvmDex.cpp:dvmDexFileOpenFromFd()函数源代码

```
int dvmDexFileOpenFromFd(int fd,DvmDex** ppDvmDex)
{
    /* 声明函数执行过程中所用到的中间变量 */
    DvmDex* pDvmDex;
    DexFile* pDexFile;
    MemMapping memMap;
    int parseFlags=kDexParseDefault;
    int result=-1;
    /* 验证 Dex 文件校验和 */
    if (gDvm.verifyDexChecksum)
        parseFlags |=kDexParseVerifyChecksum;
    if (lseek(fd,0,SEEK_SET)<0) {
        LOGE("lseek rewind failed");
        goto bail;
    }
    /* 对目标 Dex 文件进行映射,并将其设置为只读文件 */
    if (sysMapFileInShmemWritableReadOnly(fd,&memMap) !=0) {
        LOGE("Unable to map file");
        goto bail;
    }
    /* 这一步是 Dex 文件解析工作的关键,dvmDexFileOpenFromFd 函数通过调用 dexFileParse 函数对
Dex 文件进行解析,并返回一个 DexFile 数据结构的实例对象 */
  pDexFile=dexFileParse((u1 *)memMap.addr,memMap.length,
                            parseFlags);
    if (pDexFile==NULL) {
       LOGE("DEX parse failed");
       sysReleaseShmem(&memMap);
       goto bail;
    }
    /* 主函数将通过 allocateAuxStructures 函数并根据 pDexFile 变量作为参数,对 DvmDex 数据结构的
       一些成员变量进行了设置 */
    pDvmDex=allocateAuxStructures(pDexFile);
```

```
    if (pDvmDex==NULL) {
        dexFileFree(pDexFile);
        sysReleaseShmem(&memMap);
        goto bail;
    }
    sysCopyMap(&pDvmDex->memMap,&memMap);
    pDvmDex->isMappedReadOnly=true;
    *ppDvmDex=pDvmDex;
    result=0;                          //设置返回值,0表示成功
    return result;
}
```

从上面的源码中可以知道,dvmDexFileOpenFromFd 函数首先对已经经过优化的 Dex 文件进行相关的正确性检验后,随即将调用 dexFileParse 函数将目标 Dex 文件进行解析,其目标就是将该 Dex 文件与一个 DexFile 结构体对象建立关联,那么 dexFileParse 函数究竟是如何建立该联系的呢? 首先观察一下 dexFileParse 函数的源码。

代码清单 1.5　dalvik\libdex\DexFile.cpp:dexFileParse()函数源代码

```
DexFile* dexFileParse(const u1* data,size_t length,int flags)
{
    /* 声明一个 DexFile 数据结构的指针变量 pDexFile 用于保存解析结果 */
    DexFile* pDexFile=NULL;
    const DexHeader* pHeader;          //用于保存 Dex 文件的头部信息
    const u1* magic;                   //用于保存 Dex 文件的魔数信息
    int result=-1;                     //初始化返回值
    /* 对 Dex 文件大小进行判断,如果其文件长度小于其文件头的长度,则该 Dex 文件必然是错误
       的 */
    if (length< sizeof(DexHeader)) {
       LOGE("too short to be a valid .dex");
       goto bail;                      //否则为错误的格式
    }
    /* 通过 malloc 函数为 pDexFile 指针变量申请相应的 DexFile 数据结构内存空间 */
    pDexFile= (DexFile*) malloc(sizeof(DexFile));
    if (pDexFile==NULL)
         goto bail;      /* alloc failure */
    memset(pDexFile,0,sizeof(DexFile));
    /* 在对目标 Dex 文件进行解析之前,目标文件进行验证,可以看到主函数通过调用 memcmp 函数对
       Dex 文件的 magic 魔数进行验证,以明确其确为一个优化的 Dex 文件 */
    if (memcmp(data,DEX_OPT_MAGIC,4)==0) {
         magic=data;
         if (memcmp(magic+4,DEX_OPT_MAGIC_VERS,4) !=0) {
            LOGE("bad opt version (0x%02x %02x %02x %02x)",
                    magic[4],magic[5],magic[6],magic[7]);
            goto bail;
         }
    /* 将优化文件头部与 DexFile 数据结构下的 pOptHeader 变量进行关联 */
         pDexFile->pOptHeader= (const DexOptHeader*) data;
```

```
        LOGV("Good opt header,DEX offset is %d,flags= 0x%02x",
        pDexFile->pOptHeader->dexOffset,pDexFile->pOptHeader
        ->flags);
        /* 通过调用 dexParseOptData 函数对优化数据进行处理,其作用也是将各个优化数据与
      DexFile 数据结构中的相应成员变量进行关联 */
        if (!dexParseOptData(data,length,pDexFile))
            goto bail;
/* 这里可以看到,主函数通过使用 Data 变量记录当前文件所分析到的位置,用 length 记录还有多少
内容尚未分析 */
        data+ = pDexFile->pOptHeader->dexOffset;
        length -= pDexFile->pOptHeader->dexOffset;
        if (pDexFile->pOptHeader->dexLength > length) {
            LOGE("File truncated? stored len= %d,rem len= %d",
                pDexFile->pOptHeader->dexLength, (int) length);
            goto bail;
        }
        length= pDexFile->pOptHeader->dexLength;
    }
    /* 通过调用 dexFileSetupBasicPointers 函数从 data 所标示的位置继续对 Dex 文件进行
     分析,该函数的主要功能是将 Dex 文件中其他各部分数据与 DexFile 数据结构建立完整的映射关
系 */
    dexFileSetupBasicPointers(pDexFile,data);
    pHeader= pDexFile->pHeader;
    if (!dexHasValidMagic(pHeader)) {
        goto bail; }
         /* 验证 Dex 文件校验和 */
    if (flags & kDexParseVerifyChecksum) {
        u4 adler= dexComputeChecksum(pHeader);
      ⋮
        LOGV("+++adler32 opt checksum (%08x) verified",adler);
            }
        }
    }
    /* 验证 SHA-1 值 */
    if (kVerifySignature) {
        unsigned char sha1Digest[kSHA1DigestLen];
      ⋮
            LOGV("+++sha1 digest verified");
        }
    }
/* 当完成了相关的正确性检验后,设置返回标示,0 表示正确,并返回 pDexFile 变量 */
    result= 0;
    return pDexFile;
}
```

根据源码可以看到 dexFileParse 函数首先会调用 dexParseOptData 函数完成对 Odex 文件中所包含的优化数据进行分析，主要是将 DexClassLookup 哈希表以及 RegisterMaps

寄存器映射关系分别与 DexFile 数据结构下的 pClassLookup 以及 pRegisterMapPool 成员变量建立关联，随后将调用 dexFileSetupBasicPointers 函数对 Odex 文件中的原 Dex 文件进行解析，其主要工作是将 Dex 文件中各个部分数据与 DexFile 结构体部分成员变量建立指针映射，例如，将 baseAddress 指向 Dex 文件在内存映射的首地址、pHeader 指向 Dex 文件的头部、pMethodIds 指向方法区索引以及 pClassDefs 指向 Dex 文件的类数据区等。最后 dexFileParse 函数将计算校验和以及验证 SHA-1 值并正确返回，至此，dexFileParse 函数的工作结束。

点拨 哈希技术是一种典型的以空间换取时间的优化技术，这种技术可以极大地提高数据的查找效率，在大型系统中有着较为广泛的应用。哈希查找的基本原理是通过一个哈希函数将目标数据的标识符转化为一个哈希值，在哈希表中该哈希值即对应着该目标数据。简单来说，当要查找某一数据时，只需通过该哈希函数计算出目标数据的哈希值，进而再根据该哈希值在哈希表中直接取出该数据。哈希技术实际上是将常规的查找模式中所使用的一一比较的方法转化成了更加直接快速的数学计算，以此来提高查找效率。即便如此，哈希技术在实现的过程中往往需要更多的内存空间以解决哈希冲突，因此，在使用哈希技术对系统进行优化时需要考虑当前平台的硬件资源是否宽裕，否则不仅不会带来速度的提升，反而会影响系统的基本功能的正常运行。

如图 1.7 所示为 dexFileParse 函数执行流程图。

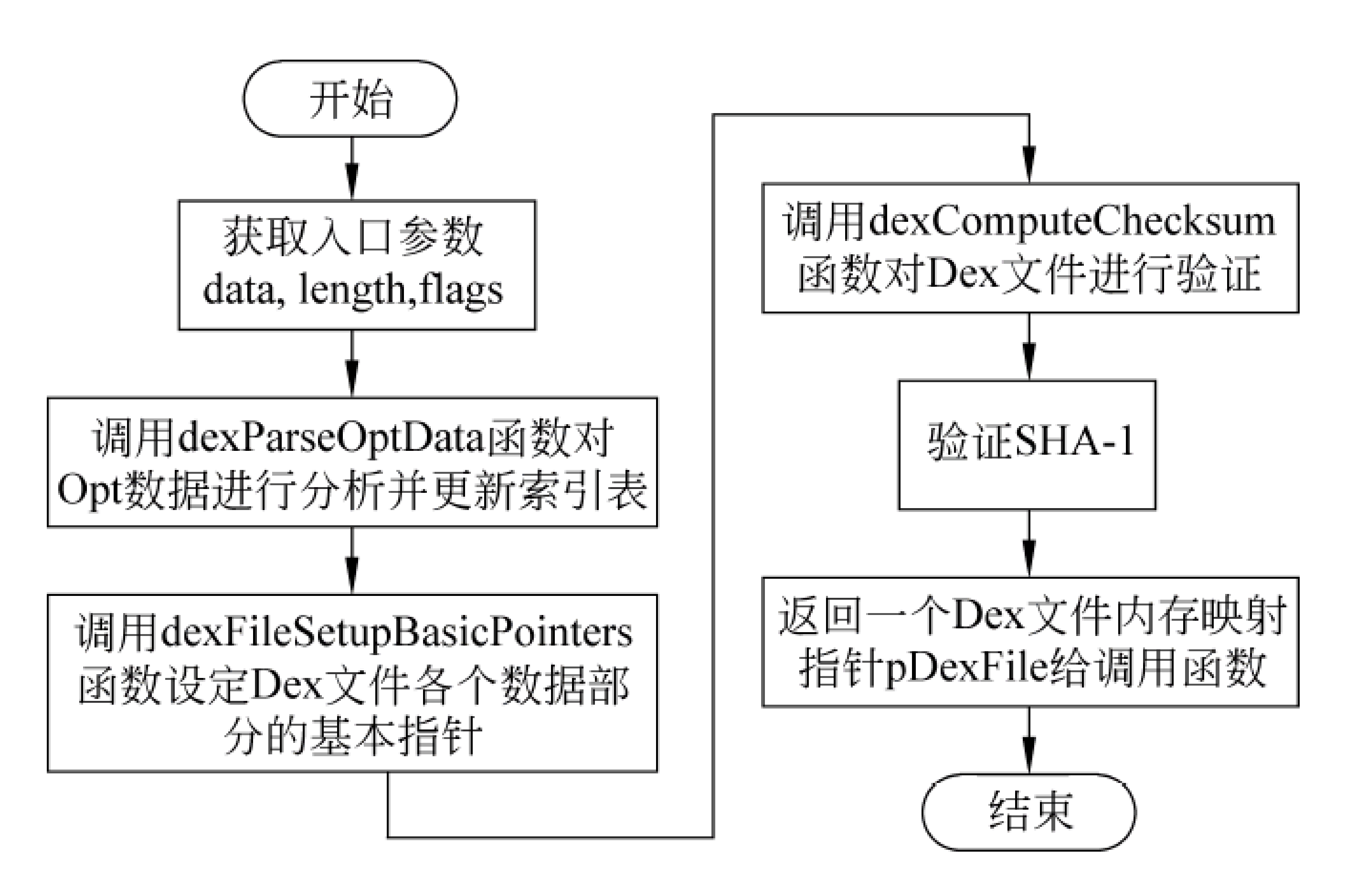


图 1.7 dexFileParse 函数执行流程图

事实上，随着 dexFileParse 函数执行的结束，Dex 文件的解析工作也即将告一段落，类加载机制接下来的工作就是根据虚拟机的运行需要，从 Dex 文件中加载指定类，并将其装入虚拟机的运行时环境中。

1.5 运行时环境数据加载

本节将对类的实际加载工作的原理与流程进行介绍，这部分的主要工作目标是将目标类的所有信息从已经解析的 Dex 文件中提取出来，并装入虚拟机的运行时环境以供解释器解释执行，这是 Dalvik 虚拟机执行流程之中非常关键的一步。

1.5.1　ClassObject 数据结构简析

类加载机制的最终目标就是为目标类生成一个 ClassObject 数据结构的实例对象，并将其存储在运行时环境中随时被执行模块引用执行。那么 ClassObject 数据结构究竟包含着哪些成员变量，而且这些成员变量都担当着怎样的职责，是本部分主要介绍的内容。首先观察一下 ClassObject 数据结构的具体定义，如表 1.5 所示。

表 1.5　struct ClassObject 结构体定义

成员变量类型	变 量 名	意　　义
const char*	descriptor	类描述符
u4	accessFlags	访问标示符
u4	serialNumber	系列号
DvmDex*	pDvmDex	指向所属 Dex 文件
ClassStatus	status	类状态标示
ClassOjbect*	verifyErrorClass	错误处理
u4	initThreadId	初始化进程 ID
ClassObject*	elementClass	元素类
int	arrayDim	数组维数
PrimitiveType	primitiveType	原始类型
ClassObject*	super	指向超类
Object*	classLoader	类装载器
int	interfaceCount	接口数目
ClassObject**	interfaces	对象接口
int	directMethodCount	直接方法数
Method*	directMethods	指向直接方法区
int	virtualMethodCount	虚方法数
Method*	virtualMethods	指向虚方法区
int	iftableCount	接口表数目
InterfaceEntry*	iftable	指向接口表
int	ifviPoolCount	常量池数量
int*	ifviPool	常量池指针
int	ifieldCount	实例字段数目
int	ifieldRefCount	引用字段数目
InstField*	ifields	实例字段指针
u4	refOffsets	字段区偏移量
Const char*	sourceFile	源文件名
int	sfieldCount	静态字段数目
StaticField	sfields	静态字段指针

从表 1.5 中可以发现，ClassObject 数据结构的成员变量数量较多，基本上包含目标类在运行期间所需要用到的全部资源，由于篇幅所限，本书在此有针对性地对几个关键的成员变量进行介绍，对于其他变量，有兴趣的读者可以结合源码进行学习，该数据结构的定义位

于 Dalvik 虚拟机源码路径下 vm/oo/Object.h 文件中。

1. pDvmDex 变量

该成员变量是一个 DvmDex 类型指针，DvmDex 数据结构实际上也是用于表示一个已被虚拟机解析的 Dex 文件，它与前面介绍的 DexFile 数据结构有何不同呢？事实上，DvmDex 数据结构只是在 DexFile 数据结构的基础上又为目标 Dex 文件增加了一些辅助信息，提高了对 Dex 文件解析的效果。

2. super 变量

该成员变量是一个 ClassObject 数据结构指针，其指向当前类的超类的 ClassObject 实例对象，通过该对象可以访问引用到其超类的相关资源。

3. classLoader 变量

该变量表示了当前类的指定加载器，一般来说，系统默认类加载器基本上可以满足应用程序中的用户类的加载工作，所以该变量往往指向该默认类加载器。事实上，类加载器也是一个类，其在内存中也有相对应的类对象实例，系统的默认类加载器是在系统启动时被装载进虚拟机。

4. directMethods 变量

该变量是一个 Method 类型的指针，作用是指向该类的直接方法区，在 1.6 节会对该数据结构进行较为详细的介绍。同时，该数据结构是虚拟机执行模块——解释器的重要输入，解释器通过该数据结构能获取欲执行方法的全部运行资源。

5. virtualMethods 变量

该变量的功能含义与 directMethods 变量非常相似，故不再赘述。

6. ifviPool 变量

该变量是一个 int 类型的指针，其作用是指向位于 Dex 文件中的数据常量池。

7. ifields 变量

该变量是一个 InstField 类型的指针，其作用是指向类的实例字段资源。

8. sfields 变量

该变量是一个 StaticField 类型的指针，其作用是指向类的静态字段资源。

以上便是 ClassObject 数据结构中至关重要的一些成员变量，事实上，这些数据结构的定义相对抽象，如果读者想要较为透彻地弄懂这些内容，作者建议读者对解释器的实际执行进行跟踪，观察虚拟机是如何通过使用这些资源以实现对一个目标程序进行解释执行的。

1.5.2　类加载整体流程概述

类加载机制的根本任务是根据程序运行需要在已被虚拟机解析的 Dex 文件中查找并加载指定类。经过 Dex 文件解析的工作后，Dex 文件中的各个部分数据对于虚拟机来说都是可见可获取的，因此在本阶段中的工作流程大致为：虚拟机在获得一个加载类的指令后，其首先确定加载类所属的 Dex 文件，随后在全局变量中查看虚拟机是否已经完成了对该 Dex 文件的解析，如果已完成解析，则返回该 Dex 文件所对应的 DexFile 数据结构，再根据欲加载类的描述符在 DexClassLookup 哈希表中查找获取目标类的各个部分数据地址，当得到 Dex 文件中相关类数据的存储地址后，将通过调用相关的加载函数对指定的各个类信息进行解析并装载，使之以 ClassObject 类型的数据结构存储于运行时环境之中，并为解释器的执行提供相应类方法的字节码。以上工作流程如图 1.8 所示。



图 1.8　类的实际加载工作流程

类加载机制最终会输出一个 ClassObject 数据结构的实例对象，该数据结构的关联关系如图 1.9 所示。

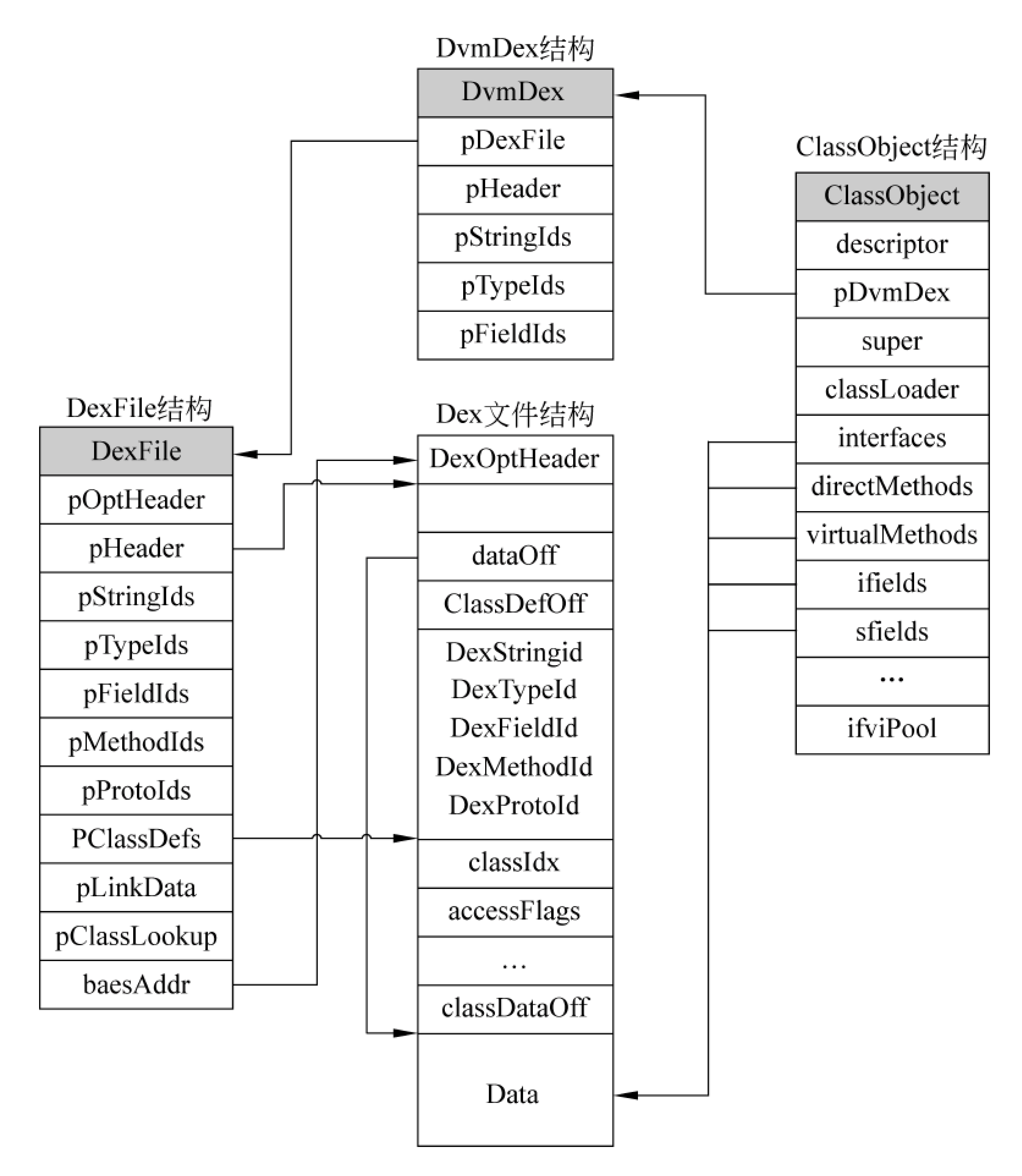


图 1.9　ClassObject 结构体结构图

从图 1.9 可以发现，虚拟机通过 ClassObject 数据结构可以获取到指定类的全部运行时数据。另外，1.6 节将会通过一个虚拟机运行实例，介绍一下 ClassObject 数据结构是如何被解释器引用并执行，以证明 ClassObject 数据结构在程序的执行过程中起到了不可替代的作用。

1.5.3 函数执行流程

Dalvik 虚拟机在本阶段的执行过程初期，通过调用类加载机制的本地方法接口函数 Dalvik_dalvik_System_DexFile_defineClass 对运行时所需的类进行定义，因此，该函数为这阶段工作中的主控函数，这阶段中的各个功能点函数在主函数的宏观调控下紧密配合共同完成了对类的实际加载工作，其函数定义位于 vm/native/dalvik_system_DexFile.cpp 文件中。

主函数 Dalvik_dalvik_system_DexFile_defineClass 在执行的初期会对入口参数进行处理，比较关键的是根据指定类名生成该类的描述符，在虚拟机中，此描述符将会作为该类的唯一标识，主函数首先调用 dvmGetRawDexFileDex 函数获取 Dex 文件在虚拟机中的 DexFile 数据结构，随后调用 dvmDefineClass 函数并以前面生成的类描述符以及相应的 DexFile 结构体指针为入口参数完成对指定类的加载工作。实际上，dvmDefineClass 函数只是调用 findClassNoInit 函数完成实际的加载工作。因此，下面主要围绕 findClassNoInit 函数的实现过程进行介绍。findClassNoInit 函数源码如下。

代码清单 1.6 dalvik\vm\oo\class.cpp：findClassNoInit()函数源代码

```
static ClassObject* findClassNoInit(const char* descriptor,Object* loader,
    DvmDex* pDvmDex)
{
    /* 声明一些中间变量 */
    Thread* self=dvmThreadSelf();
    ClassObject* clazz;                //变量 clazz 为类加载的最终表现形式
    bool profilerNotified=false;
    /* 判断目标类是否有类加载器,事实上,对于系统类,虚拟机将从默认的启动路径实现其加载工
     作;对于用户类,虚拟机一般情况下使用默认的类加载器实现加载工作 */
    if (loader !=NULL) {
       LOGVV("####findClassNoInit(%s,%p,%p)",descriptor,loader,
            pDvmDex->pDexFile);
    }
  ⋮
    /* 根据目标类的描述符 descriptor 在系统已加载类中进行查找,如果已对其加载,则返回目标类
的 ClassObject 对象;否则,将对目标类进行加载 */
    clazz=dvmLookupClass(descriptor,loader,true);
    if (clazz==NULL) {
        const DexClassDef* pClassDef;
        dvmMethodTraceClassPrepBegin();
        profilerNotified=true;
/* 判断是否存在 DvmDex 结构体对象,如果存在,则表示目标类为一个用户类,我们将要从一个解析的 Dex
文件中进行加载,对于一个解析过的 Dex 文件,是一定存在一个 DvmDex 结构体对象的,故 pDvmDex 一定不为
```

空;若为空则表示目标类是一个系统类,虚拟机将调用 searchBootPathFo rClass 函数从启动路径下查找并加载目标类 * /

```
        if (pDvmDex==NULL) {
            assert(loader==NULL);      /* shouldn't be here otherwise */
            pDvmDex=searchBootPathForClass(descriptor,&pClassDef);
        } else {
```

/* 在这里,可以看到主函数调用 dexFindClass 并以 DexFile 数据结构的实例对象 pDexFile 作为参数,查找目标类的类定义资源,并将结果返回给 pClassDef 变量 * /

```
            pClassDef=dexFindClass(pDvmDex->pDexFile,descriptor);
        }
        ⋮
```

/* 当获得了加载目标类所需的各项资源,主函数将调用 loadClassFromDex 函数对目标类进行加载 * /

```
        clazz=loadClassFromDex(pDvmDex,pClassDef,loader);
        if (dvmCheckException(self)) {
            if (clazz !=NULL) {
                dvmFreeClassInnards(clazz);
                dvmReleaseTrackedAlloc((Object*) clazz,NULL);
            }
            goto bail;
        }
        /* 将目前使用的类锁住,防止其他进程更改 */
        dvmLockObject(self,(Object*) clazz);
        clazz->initThreadId=self->threadId;
        /* 增加到哈希表中 */
        assert(clazz->classLoader==loader);
        if (!dvmAddClassToHash(clazz)) {
            clazz->initThreadId=0;
            dvmUnlockObject(self,(Object*) clazz);
            /* 将中间变量 clazz 释放 */
            dvmFreeClassInnards(clazz);
            dvmReleaseTrackedAlloc((Object*) clazz,NULL);
            /* 从已加载类的系统哈希表中重新得到的类 */
            clazz=dvmLookupClass(descriptor,loader,true);
            assert(clazz !=NULL);
            goto got_class;
        }
        dvmReleaseTrackedAlloc((Object*) clazz,NULL);
        /* 准备开始连接类 */
        if (!dvmLinkClass(clazz)) {
            assert(dvmCheckException(self));
            /* 记录错误并且将类清空 */
            removeClassFromHash(clazz);
            clazz->status=CLASS_ERROR;
            dvmFreeClassInnards(clazz);
```

```
                clazz->initThreadId=0;
                dvmObjectNotifyAll(self,(Object*) clazz);
                dvmUnlockObject(self,(Object*) clazz);
                 clazz->descriptor,get_process_name());
          ⋮
            /*将类的状态增加到全局变量中去*/
            gDvm.numLoadedClasses++;
            gDvm.numDeclaredMethods+=
                clazz->virtualMethodCount+clazz->directMethodCount;
            gDvm.numDeclaredInstFields+=clazz->ifieldCount;
            gDvm.numDeclaredStaticFields+=clazz->sfieldCount;
                ⋮
        /*对一些变量进行检查*/
            assert(dvmIsClassLinked(clazz));
            assert(gDvm.classJavaLangClass !=NULL);
            assert(clazz->clazz==gDvm.classJavaLangClass);
            assert(dvmIsClassObject(clazz));
        assert(clazz==gDvm.classJavaLangObject || clazz->super !=NULL);
        if (!dvmIsInterfaceClass(clazz)) {
            //LOGI("class=%s vtableCount=%d,virtualMeth=%d",
            //      clazz->descriptor,clazz->vtableCount,
            //      clazz->virtualMethodCount);
            assert(clazz->vtableCount >=clazz->virtualMethodCount);
        }
    /*错误处理*/
    bail:
        if (profilerNotified)
            dvmMethodTraceClassPrepEnd();
        assert(clazz !=NULL || dvmCheckException(self));
        return clazz;
    }
```

从上面的源码可以看到，findClassNoInit 函数负责对一个指定类进行实际加载的工作，在其执行的初期，会先调用 dvmLookupClass 函数根据类的描述符在全局变量 gDvm. loadedClasses（该全局变量记录了当前虚拟机加载过的所有类）中进行查找并判断目标类是否已被加载过，如果已经加载，那么直接引用这个已加载类并将类对象指针返回给调用函数，否则将对其进行加载，如图 1.10 所示。

事实上，被加载的类实际上分为两种：

（1）系统基本类；

（2）用户类。

对这两种类进行加载时的主要区别体现在 Dex 文件查找方式的不同，当欲加载的类为系统基本类时，findClassNoInit 函数通过调用 searchBootPathForClass 函数从系统启动基本路径中查找并加载目标类；在查找用户类时，主控函数将会调用 dexFindClass 函数根据

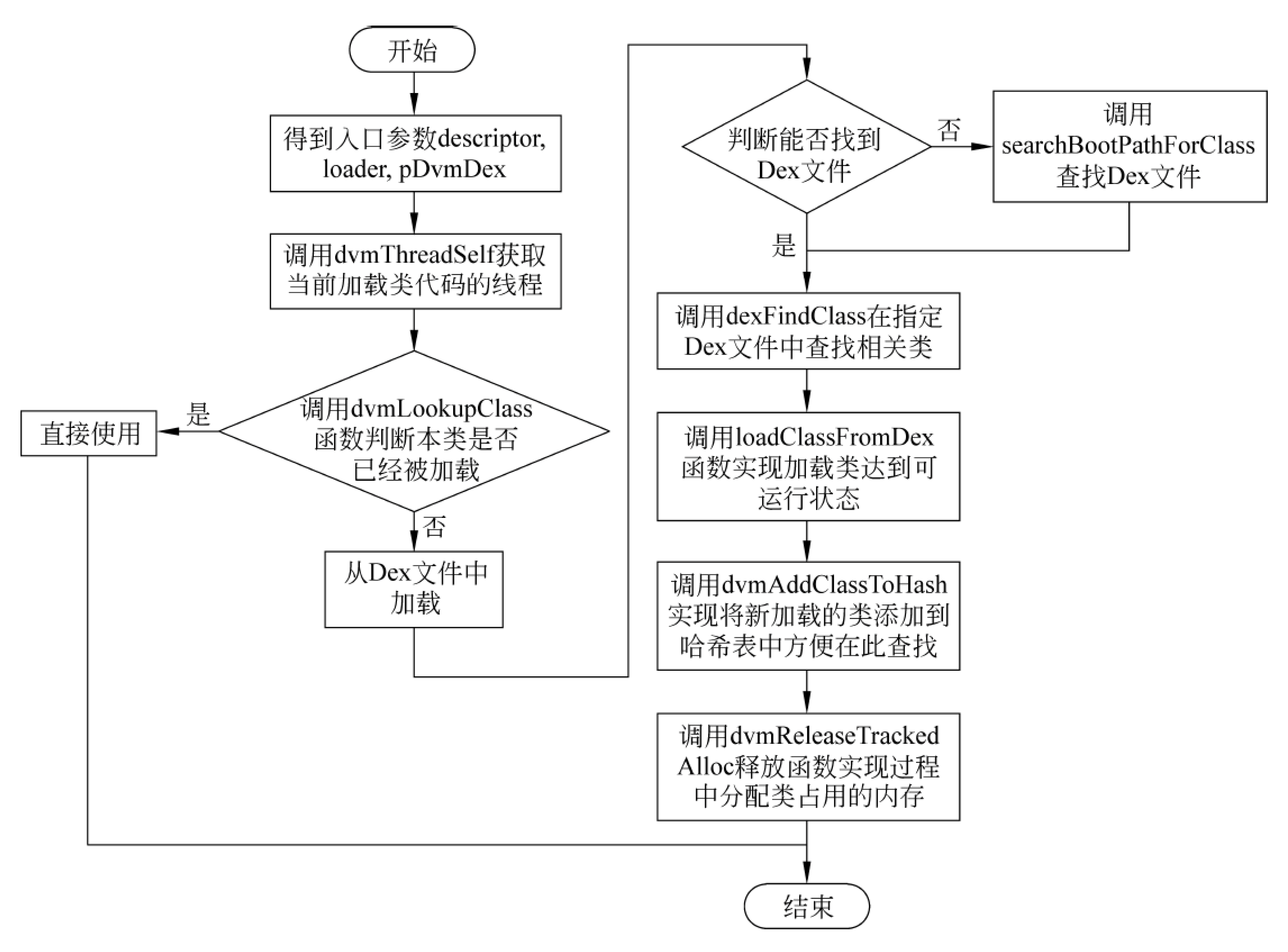


图 1.10　findClassNoInit 函数执行流程图

类的描述符在 Odex 文件的类索引表中进行查找匹配，该函数的返回为一个 DexClassDef 结构体对象，该结构体定义如表 1.6 所示。

表 1.6　DexClassDef 数据结构定义

变量类型	变量名称	描　　述
u4	classIdx	类索引
u4	accessFlags	访问标示符
u4	superclassIdx	超类索引
u4	interfaceOff	接口数据偏移量
u4	sourceFileIdx	源文件索引
u4	annotationIdx	元数据信息索引
u4	classDataOff	类数据偏移量
u4	StaticValueOff	静态数据偏移量

从表 1.6 中可以看到，虚拟机通过该数据结构可以快速定位目标类各个部分数据位于 Dex 文件中的位置，在一定程度上为后续加载函数提供了便利。当 dexFindClass 函数正确返回后，findClassNoInit 函数将调用 loadClassFormDex0 函数完成对该类的加载工作。loadClassFormDex0 函数的返回值为一个 ClassObject 结构体对象，这表明该函数将对目标类进行实际加载。loadClassFormDex0 函数的源码如下。

代码清单 1.7 dalvik\vm\oo\class.cpp:loadClassFormDex0()函数源代码

```
static ClassObject* loadClassFromDex0(DvmDex* pDvmDex,
    const DexClassDef* pClassDef,const DexClassDataHeader* pHeader,
    const u1* pEncodedData,Object* classLoader)
{
    /*声明相关的中间变量*/
    ClassObject* newClass=NULL;              //目标类的类实例对象
const DexFile* pDexFile;              //用于储存目标 Dex 文件所对应的 DexFile 数据结构实例对象
const char* descriptor;                  //用于储存目标类的描述符
    /*获取相应的类信息*/
    pDexFile=pDvmDex->pDexFile;
    descriptor=dexGetClassDescriptor(pDexFile,pClassDef);
⋮
    /*为即将生成的类对象实例申请内存空间*/
    assert(descriptor !=NULL);
    if (classLoader==NULL &&
       strcmp(descriptor,"Ljava/lang/Class;")==0) {
       assert(gDvm.classJavaLangClass !=NULL);
       newClass=gDvm.classJavaLangClass;
    } else {
    /*取得对象实例大小并在内存中申请相应内存*/
         size_t size=classObjectSize(pHeader->staticFieldsSize);
  newClass=(ClassObject*) dvmMalloc(size,ALLOC_NON_MOVING);
    }
    if (newClass==NULL)
       return NULL;
    /*对新的类对象实例进行初始化*/
    DVM_OBJECT_INIT(newClass,gDvm.classJavaLangClass);
    dvmSetClassSerialNumber(newClass);
    newClass->descriptor=descriptor;
    assert(newClass->descriptorAlloc==NULL);
    SET_CLASS_FLAG(newClass,pClassDef->accessFlags);
    /*设定字段对象*/
    dvmSetFieldObject((Object *)newClass,OFFSETOF_MEMBER(ClassO
                 bject,classLoader),(Object *)classLoader);
    /*设定类的相关指针*/
    newClass->pDvmDex=pDvmDex;
    newClass->primitiveType=PRIM_NOT;
    newClass->status=CLASS_IDX;
     /*将这个类的父类的索引加入到类对象的指针区域*/
    assert(sizeof(u4)==sizeof(ClassObject*)); /* 32-bit check */
    newClass->super=(ClassObject*) pClassDef->superclassIdx;
     /*设置类的参考指针*/
    const DexTypeList* pInterfacesList;
    /*得到接口列表*/
```

```
pInterfacesList=dexGetInterfacesList(pDexFile,pClassDef);
if (pInterfacesList !=NULL) {
 /* 得到接口数目 */
     newClass->interfaceCount=pInterfacesList->size;
     /* 得到接口 */
 newClass->interfaces=(ClassObject**) dvmLinearAlloc(classLoa
           der,newClass->interfaceCount * sizeof(ClassObject*));
     /**逐一对接口进行处理 */
     for (i=0; i<newClass->interfaceCount; i++) {
     const DexTypeItem* pType=dexGetTypeItem(pInterfacesList,i);
     newClass->interfaces[i]=(ClassObject*)(u4) pType->typeIdx;
     }
     dvmLinearReadOnly(classLoader,newClass->interfaces);
}
 /* 对字段进行加载,首先加载静态字段 */
 if (pHeader->staticFieldsSize !=0) {
     int count=(int) pHeader->staticFieldsSize;
     u4 lastIndex=0;
     DexField field;
     /* 取得字段数 */
     newClass->sfieldCount=count;
     /* 逐一加载字段 */
     for (i=0; i<count; i++) {
     dexReadClassDataField(&pEncodedData,&field,&lastIndex);
     loadSFieldFromDex(newClass,&field,&newClass->sfields[i]);
     }
}
/* 加载实例字段 */
if (pHeader->instanceFieldsSize !=0) {
     int count=(int) pHeader->instanceFieldsSize;
     u4 lastIndex=0;
     DexField field;
     /* 取得字段数 */
     newClass->ifieldCount=count;
     newClass->ifields=(InstField*) dvmLinearAlloc(classLoader,
             count * sizeof(InstField));
     /* 逐一加载字段 */
     for (i=0; i<count; i++) {
      dexReadClassDataField(&pEncodedData,&field,&lastIndex);
      loadIFieldFromDex(newClass,&field,&newClass->ifields[i]);
     }
     dvmLinearReadOnly(classLoader,newClass->ifields);
}
 …
/* 对类方法进行加载 */
```

```
if (pHeader->directMethodsSize != 0)          //判断直接方法数是否为 0
{
        int count= (int) pHeader->directMethodsSize;
        u4 lastIndex= 0;
        DexMethod method;
        /* 取得方法数目 */
    newClass->directMethodCount= count;
    newClass->directMethods= (Method* ) dvmLinearAlloc(classLoader,
                count *  sizeof(Method));
        /* 逐一加载方法 */
        for (i= 0; i< count; i++) {
    dexReadClassDataMethod(&pEncodedData,&method,&lastIndex);
loadMethodFromDex(newClass,&method,&newClass->directMethods[i]);
            if (classMapData != NULL) {
  const RegisterMap*  pMap= dvmRegisterMapGetNext(&classMapData);
  if (dvmRegisterMapGetFormat(pMap) != kRegMapFormatNone) {
            newClass->directMethods[i].registerMap= pMap;
        assert((newClass->directMethods[i].registersSize+ 7)/8
         ==newClass->directMethods[i].registerMap->regWidth);
                }
            }
        }
        dvmLinearReadOnly(classLoader,newClass->directMethods);
    }
  /* 加载虚方法 */
  if (pHeader->virtualMethodsSize != 0)       //判断虚方法数是否为 0
{
        int count= (int) pHeader->virtualMethodsSize;
        u4 lastIndex= 0;
        DexMethod method;
        /**取得方法数目 */
    newClass->virtualMethodCount= count;
    newClass->virtualMethods= (Method* ) dvmLinearAlloc(classLoader,
                count *  sizeof(Method));
        /**逐一处理方法 */
        for (i= 0; i< count; i++) {
  dexReadClassDataMethod(&pEncodedData,&method,&lastIndex);
loadMethodFromDex(newClass,&method,&newClass->virtualMethods[i]);
  if (classMapData != NULL) {
  const RegisterMap*  pMap= dvmRegisterMapGetNext(&classMapData);
     if (dvmRegisterMapGetFormat(pMap) != kRegMapFormatNone) {
         newClass->virtualMethods[i].registerMap= pMap; assert ((newClass->virtualMethods[i].
         registersSize+ 7)/8== newClass->virtualMethods[i].registerMap->regWidth);
                }
            }
```

```
            }
            dvmLinearReadOnly(classLoader,newClass->virtualMethods);
    }
    /* 保存源文件信息 */
        newClass->sourceFile= dexGetSourceFile(pDexFile,pClassDef);
        return newClass;                    //返回类对象
    }
```

loadClassFormDex0 函数作为实际加载工作的承担者，实际上，其工作逻辑相对简单，即依次完成：①在内存中为类对象申请存储空间；②设置字段信息；③为超类建立索引；④加载类接口；⑤加载类字段；⑥加载类方法，并将以上数据封装成一个 ClassObject 结构体对象并返回。

当 loadClassFormDex0 函数正常返回后，findClassNoInit 函数要对全局变量 gDvm.loadedClasses 哈希表进行更新以及进行类的连接工作。至此，类加载机制的全部工作结束。

如图 1.11 所示为 loadClassFormDex0 函数执行流程图。

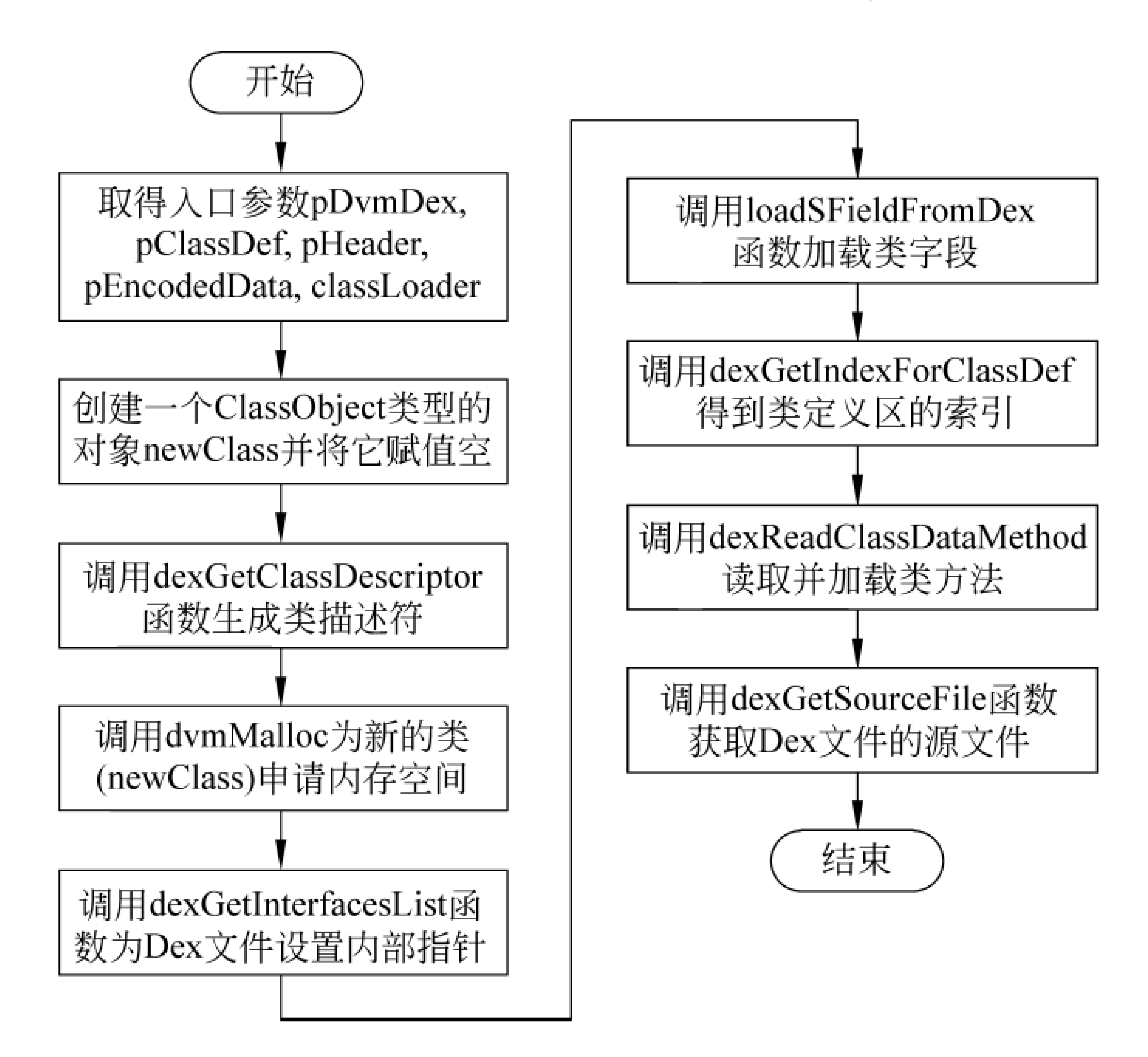


图 1.11　loadClassFormDex0 函数执行流程图

1.6　类加载机制与解释器交互示例

ClassObject 数据结构作为程序类被加载后的表现形式，同时它还是解释器机制的重要输入，有必要讨论一下该数据结构是如何承担起程序正确运行的重要作用，本文在此通过一个程序运行实例介绍一下解释器是如何通过使用 ClassObject 对象中的资源去解释执行一段程序。

对于一个程序的执行，当虚拟机完成类的加载工作后，解释器会对已装载的代码进行检

查，验证是否符合虚拟机的格式规范，最后调用 dvmInterpret 函数初始化解释器并开始解释执行字节码，dvmInterpret 函数是解释器的入口函数，接下来宏观观察一下这个函数：dvmInterpret(Thread * self, const Method * method, JValue * pResult)，可以发现这个函数的第二参数是一个 Method 类型的指针，然而这个指针正是 ClassObject 的一个成员变量，由此可以看到这是类加载器和解释器的一个关键的交汇点，但这还不是最核心的内容，先观察一下 Method 类型结构体的细节，如表 1.7 所示。

表 1.7 Method 数据结构定义

数据类型	变量名	变 量 含 义
ClassObject*	clazz	一个类对象实例
u4	accessFlags	访问标示符
u2	methodIndex	方法索引
u2	registersSize	使用寄存器数量
u2	outsSize	outs 单元大小
u2	insSize	ins 单元大小
const char*	name	方法名
DexProto	prototype	原类型
const char*	shorty	短格式的方法描述字符串
const u2*	insns	实际代码，Dex 文件在内存中的映射的方法区
int	jniArgInfo	jni 信息
DalvikBridgeFunc	nativeFunc	本地方法引用
bool	fastJni	标示本地方法是否需要一个 JNIEnv* 或者 jclass
bool	noRef	引用标示
bool	shouldTrace	是否需要被日志所记录
const RegisterMap*	registerMap	寄存器映射
bool	inProfile	在 profiling 时这个方法是否被调用

表 1.7 中的第一个参数 clazz，它实际是一个 ClassObject 类型的指针，它指向这个方法隶属于哪个 ClassObject 实例，然而在这个结构体中最关键的成员变量是一个 const u2* 类型的指针变量 insns，它指向了 Dex 文件的实际可运行代码区，这个变量将在解释器入口函数 dvmInterpret 中有重要体现，下面是 dvmInterpret 函数的一段关键代码。

```
/** 初始化解释器工作环境 * /
self->interpSave.method=method;
self->interpSave.curFrame= (u4 * ) self->interpSave.curFrame;
self->interpSave.pc=method->insns;
```

从上述代码中可以看到，这个执行进程 self 首先取得了将要执行方法的 method 指针，随后在第三行代码中，虚拟机将 method→insns 可执行代码区的首地址赋值给了执行进程的 pc 指针 self→interpSave. pc，随后解释器的 pc 指针将会逐一读入指令并执行相关字节码。

小　　结

本章主要以类加载机制的设计原理以及具体实现为研究重点，对类加载机制整体功能架构设计、实现原理、机制执行流程以及机制内部函数调用关系进行了细致的研究，并结合一个虚拟机运行实例描述了类加载机制产物在虚拟机运行过程中所起的重要作用。同时，本书还从实现原理以及工作流程两方面对 Dex 文件的优化机制进行了阐述，该机制作为 Dalvik 虚拟机与标准 Java 虚拟机重要的区别体现，是 Dalvik 类加载机制最关键的功能模块之一，对类加载机制乃至 Dalvik 虚拟机整体性能都有着重要意义，更重要的是为后续在低性能的嵌入式平台上研究实现高性能 Java 语言虚拟机提供了重要参考。

第 2 章 内存管理的原理及实现

本章主要内容

- ✍ 内存管理机制中涉及的关键数据结构有哪些？
- ✍ 内存管理机制中涉及的关键函数有哪些？
- ✍ 内存分配的算法和流程是怎样的？
- ✍ 当前主要的垃圾回收算法有哪些？
- ✍ 垃圾回收的流程是怎样的？

为了保证 Android 系统的正常运行和应用程序的稳定性，Dalvik 虚拟机的内存管理机制在整个虚拟机系统中占有非常重要的位置。Dalvik 虚拟机采用怎样的算法和流程来分配内存空间？对于空闲不再使用的内存又是采用怎样的算法和流程来回收？本章针对以上疑问一一解答，从源代码入手，分析 Dalvik 虚拟机内存管理的原理和实现。

2.1 内存管理初探

内存分配和垃圾回收机制是 Dalvik 虚拟机整体设计实现架构中的一个关键部分，用户并不显式地进行垃圾回收工作，所以，为了保证 Dalvik 虚拟机的正常运行以及 Android 系统的快速和稳定性，必须存在一套行之有效的机制来保证内存分配和垃圾回收工作的顺利进行，内存管理部分与其他模块之间的关系如图 2.1 所示。

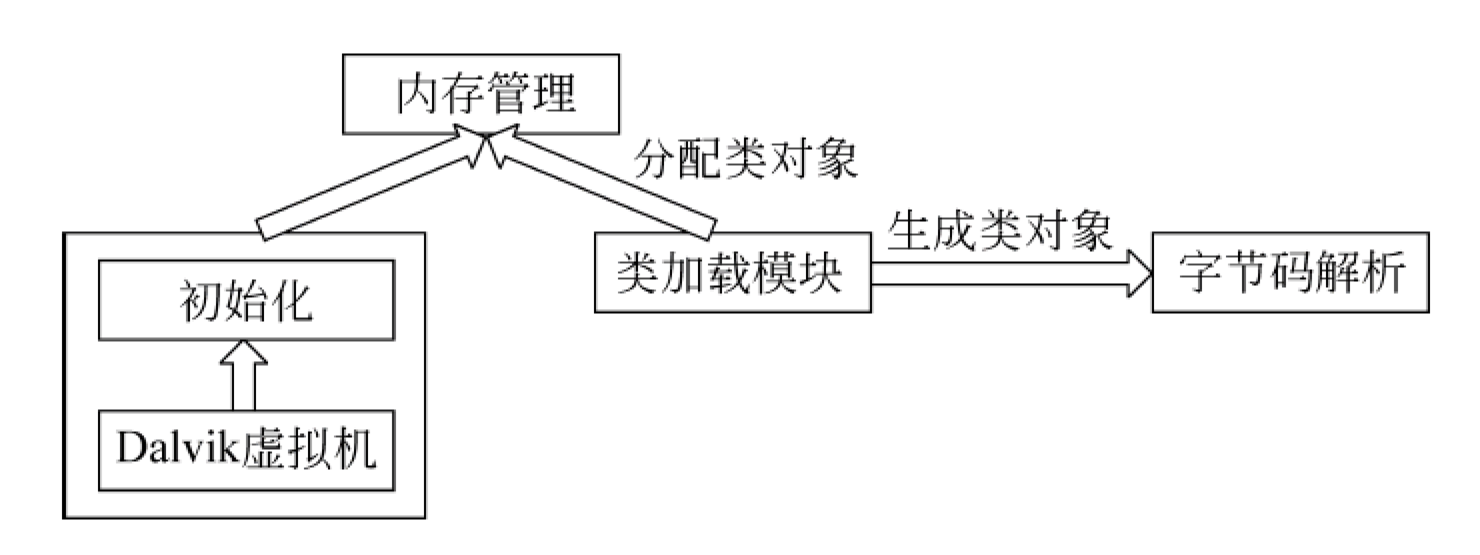


图 2.1 内存管理部分与其他模块关系

Dalvik 虚拟机内存分配的底层依赖是基于 Doug Lea 编写的 dlmalloc 内存分配器，在 Heap 上完成，按照分配规则，每分配一个内存区域经过数次尝试，如果第一次内存分配失败了，那么进行一次垃圾收集，这次垃圾收集并不收集软引用对象，收集完成之后，再次尝试进行内存分配，如果第二次尝试再次失败，就增长堆的大小，并进行第三次的分配尝试，因为堆是可以在堆生长限制之内进行生长的。如果第三次尝试再次失败，启动垃圾收集器

进行垃圾收集，此次收集过程收集软引用对象，收集过程完成之后进行第四次的内存分配尝试，如果分配成功则返回一个指向内存区域的指针，否则，如果失败返回空指针并且抛出异常，虚拟机暂停工作。在整个内存分配尝试中，进行了两次垃圾收集，第一次不收集 SoftReference，第二次收集 SoftReference。垃圾收集的目的就是及时回收那些不再使用的对象的空间，保证内存堆中有足够的空间可以分配对象，分配过程中最多经过 4 次尝试。

点拨　详细了解 Doug Lea 编写的 dlmalloc 内存分配器的源代码，可以访问 http://fayaa.com/code/view/3719/。

Dalvik 虚拟机在垃圾回收时使用 Mark Sweep 算法，该算法一般分为 Mark 阶段和 Sweep 阶段。Mark 阶段就是标记出活动对象，使用栈来保存根集合，然后对栈中的每一个元素，递归追踪所有可访问的对象，对于所有可访问的对象，在 markBits 位图中将该对象的内存起始地址对应的位设为 1。这样当栈为空时，markBits 位图就是所有可访问的对象集合。垃圾收集的第二步就是回收内存，在 Mark 阶段通过 markBits 位图可以得到所有可访问的对象集合，而 liveBits 位图表示所有已经分配的对象集合。因此通过比较这两个位图，liveBits 位图和 markBits 位图的差异就是所有可回收的对象集合。内存管理部分的框图如图 2.2 所示。

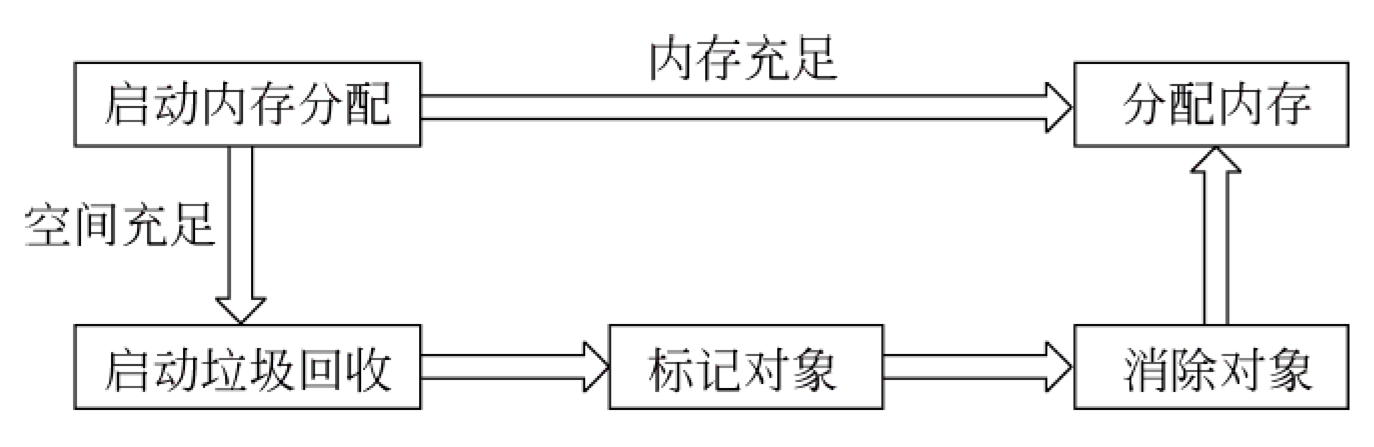


图 2.2　内存管理部分框图

Dalvik 虚拟机的内存分配和回收是在 HeapSource 结构体上进行的，而每一个 HeapSource 结构体又包含多个 Heap 结构体成员，确切地说，内存管理的场所是 Heap，在 Heap 上分配内存空间或者回收不再使用的内存空间。HeapBitmap 结构体用于内存回收，保存了对已分配内存空间的标记，垃圾回收时将没有被标记的内存回收。本章下文将详细介绍各个结构体及其成员变量。

Dalvik 虚拟机的内存管理机制的实现涉及许多函数，本章选取一些在实现过程中起关键作用的关键函数进行分析。dvmAllocObject 函数的主要功能是虚拟机为对象分配内存空间；dvmHeapMarkRootSet 函数用于垃圾回收的标记过程；dvmHeapScanMarkedObjects 函数主要实现给定根标记的位图，找到并且标记所有可触及的对象；processMarkStack 函数是在 Dalvik 虚拟机垃圾回收的标记步骤中用到的处理标记栈的关键函数；ptr2heap 函数返回指定对象的堆；createMspace 函数创建一个不是锁定状态的 dlmalloc 内存空间作为一个堆资源；allocMarkStack 函数是分配内存空间的关键函数；dvmHeapSourceAlloc 函数是 Dalvik 虚拟机内存分配部分负责分配内存空间的关键函数，分配过程是在 HeapSource 上的底层内存资源 Heap 上进行的。

2.2 内存分配过程分析

2.2.1 关键数据结构

1. HeapSource 结构体

HeapSource 结构体是在 dalvik/vm/HeapSource.cpp 中定义的，以下为 HeapSource 结构体的实现代码和各个成员变量的含义。

代码清单 2.1 dalvik/vm/HeapSource.cpp：HeapSource 结构体源代码

```
struct HeapSource {
    size_t targetUtilization;          /**理想的堆利用率 * /
    size_t startSize;                  /**初始堆大小 * /
    size_t maximumSize;                /**堆资源作为整体允许生长的最大大小 * /
    size_t growthLimit;                /**堆可以生长到的最大大小 * /
    size_t idealSize;                  /**堆资源作为整体的理想最大值 * /
    size_t softLimit;                  /**允许从活动的堆中分配的最大字节数 * /
/**<heap[0]通常是活动的堆,新的对象应该从这里分配空间 * /
    Heap heaps[HEAP_SOURCE_MAX_HEAP_COUNT];
    size_t numHeaps;                   /**当前堆的数量 * /
    bool sawZygote;                     /**如果 HeapSource 创建的时候 zygote 是活动的,则为真 * /
    char * heapBase;                   /**内存的基地址  * /
    size_t heapLength;                 /**内存字节长度 * /
    HeapBitmap liveBits;               /**存活的对象位图 * /
    HeapBitmap markBits;               /**标记位图 * /

    /**GC 后台程序的状态量.* /
    bool hasGcThread;
    pthread_t gcThread;
    bool gcThreadShutdown;
    pthread_mutex_t gcThreadMutex;
    pthread_cond_t gcThreadCond;
    bool gcThreadTrimNeeded;
};
```

2. Heap 结构体

Heap 结构体在 dalvik/vm/HeapSource.cpp 中定义，以下为结构体的实现代码和各个成员变量的含义。

代码清单 2.2 dalvik/vm/HeapSource.cpp：Heap 结构体源代码

```
struct Heap {
    mspace msp;                        /**内存分配的源内存空间 * /
    size_t maximumSize;                /**堆可以生长到的最大大小 * /
    size_t bytesAllocated;             /**从内存空间中给对象分配的内存大小 * /
```

```
    size_t concurrentStartBytes;        /**并发垃圾收集开始前分配的内存大小 * /
    size_t objectsAllocated;            /**当前从内存空间中分配的对象数量 * /
    char * base;                        /**堆的最低地址 * /
    char * limit;                       /**堆的最高地址 * /
};
```

Dalvik 虚拟机维护了两个对应于内存的位图，分别为 liveBits 和 markBits。liveBits 标记着所有已经分配内存空间的对象，而 markBits 标记着根节点和所有正在使用的对象。这样，所有在 liveBits 中标识而没在 markBits 中标识的位对应的对象就是垃圾，应该在垃圾回收时被回收释放。

点拨　Heap 是一个和底层关系很近的数据结构，对象的内存分配和释放直接在这上面进行。

2.2.2　关键函数

1. dvmAllocObject 函数

dvmAllocObject 函数在 dalvik/vm/alloc/Alloc.cpp 文件中定义，源代码如下所示，函数流程图如图 2.3 所示。

代码清单 2.3　dalvik/vm/alloc/Alloc.cpp：dvmAllocObject()源代码

```
Object * dvmAllocObject(ClassObject * clazz,int flags)
{
    Object * newObj;                    //声明新的对象
    assert(clazz !=NULL);
    assert(dvmIsClassInitialized(clazz) || dvmIsClassInitializing(clazz));

    newObj= (Object * )dvmMalloc(clazz->objectSize,flags);
    if (newObj !=NULL) {                //如果对象不为空
        DVM_OBJECT_INIT(newObj,clazz);
        dvmTrackAllocation(clazz,clazz->objectSize);
    }

    return newObj;                      //返回分配的对象
}
```

首先获得入口参数 clazz 和 flags。定义 newObj 指针。然后在垃圾回收堆上进行内存分配，实现 newObj＝(Object *) dvmMalloc(clazz→objectSize, flags)。判断 newObj 是否为 null，若不为 null，则执行函数，进行 DVM 对象初始化，通知调试监控服务。若为 null，则直接返回 newObj。

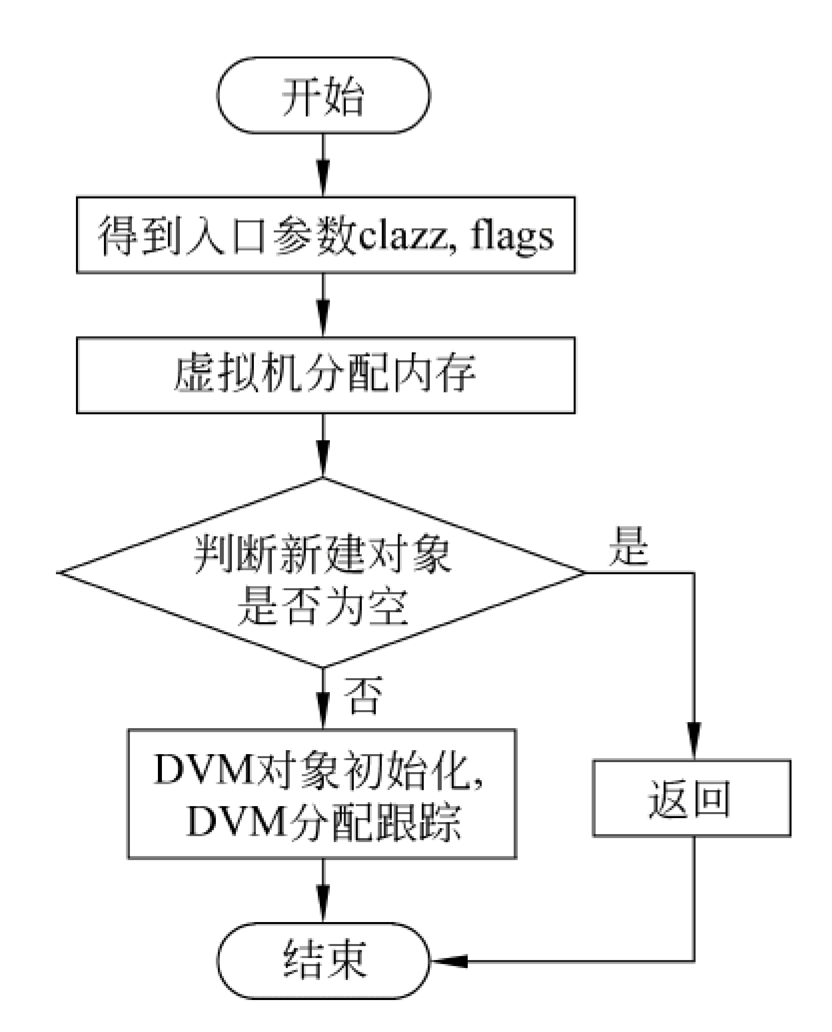


图 2.3　dvmAllocObject 函数流程图

2. createMspace 函数

createMspace 在 dalvik/vm/alloc/HeapSource.cpp

中定义，源代码如下所示，函数流程图如图 2.4 所示。

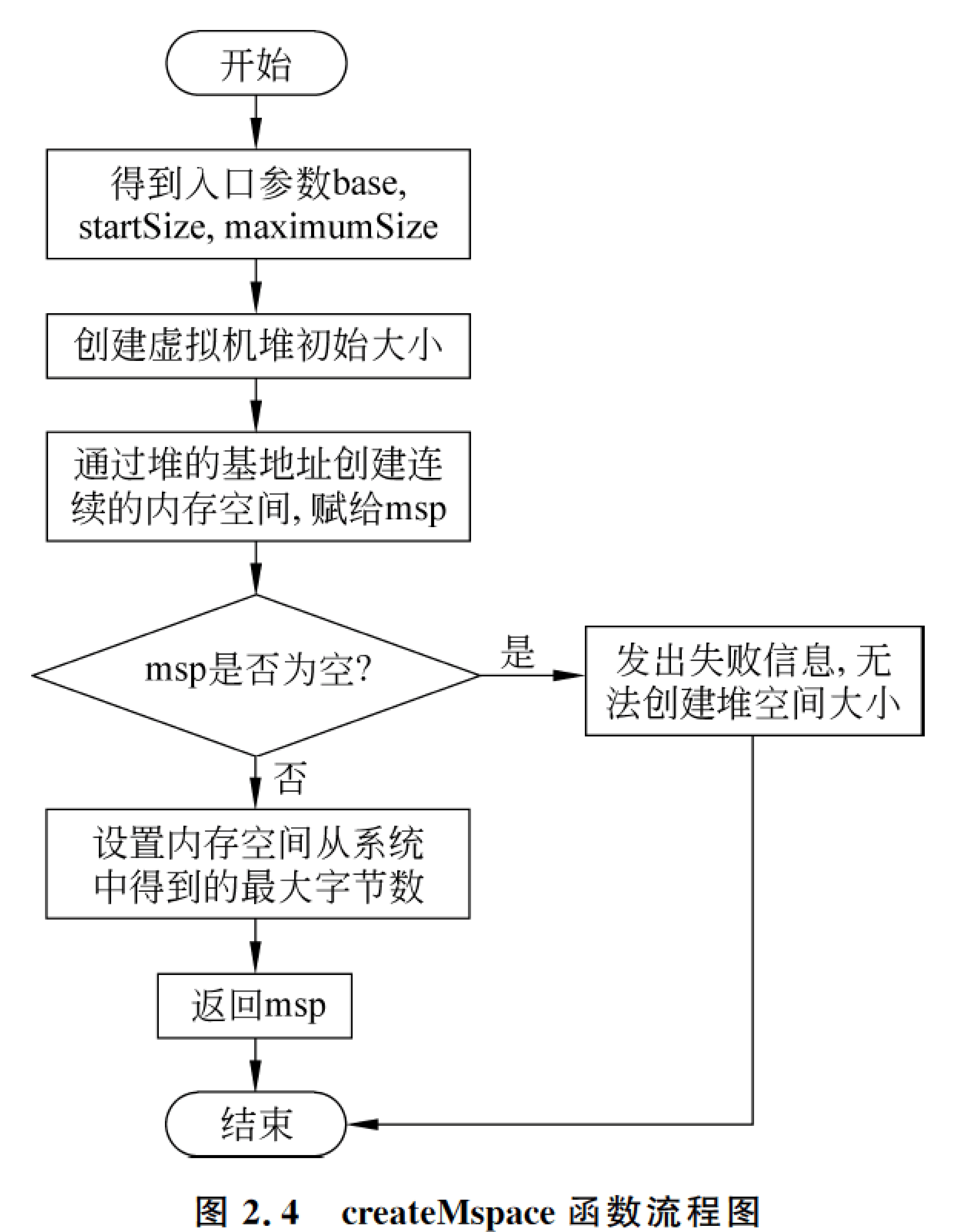


图 2.4 createMspace 函数流程图

代码清单 2.4 dalvik/vm/alloc/HeapSource.cpp：createMspace()源代码

```
static mspace createMspace(void * base,size_t startSize,size_t maximumSize)
{

    LOGV_HEAP("Creating VM heap of size %zu",startSize);
    errno= 0;

    mspace msp= create_contiguous_mspace_with_base(startSize/2,
            maximumSize,/* locked= * /false,base);          //创建连续的内存区域
    if (msp != NULL) {                                       //如果分配成功

                mspace_set_max_allowed_footprint(msp,startSize);
    } else {

        LOGE_HEAP("Can't create VM heap of size (%zu,%zu): %s",
            startSize/2,maximumSize,strerror(errno));
    }

    return msp;                                              //返回分配的内存区域
}
```

createMspace 函数的主要功能是，创建一个不是锁定状态的 dlmalloc 内存空间作为一

个堆，开始保留 startSize/2 比特但是允许它生长到 startSize。首先得到入口参数，base 为创建堆的基地址、startSize 为创建堆的初始大小、maximumSize 为创建堆的最大大小。然后创建虚拟机堆初始大小。通过堆的基地址创建连续的空间，赋给 msp。create_contiguous_mspace_with_base 函数有 startSize/2，maximumSize，false，base4 个参数。接着判断 msp 是否为 null，若为 null，则发出失败信息，无法创建堆空间大小，接着结束；若不为空，则设置内存空间为从系统得到的最大字节数，返回 msp，最后结束。

3. addNewHeap 函数

addNewHeap 在 dalvik/vm/alloc/HeapSource.cpp 中定义，源代码如下所示，函数流程图如图 2.5 所示。

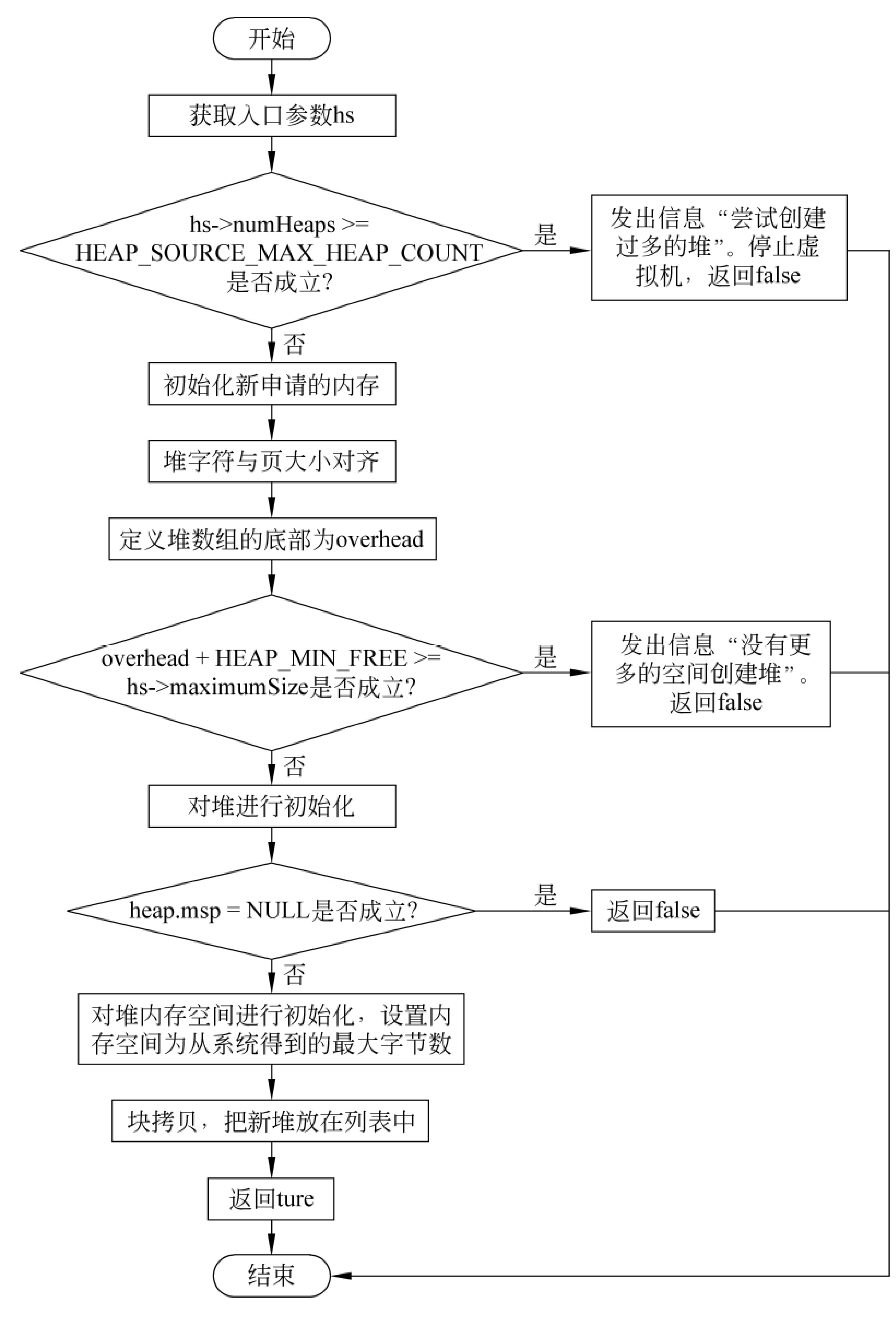


图 2.5　addNewHeap 函数流程图

代码清单 2.5 dalvik/vm/alloc/HeapSource.cpp：addNewHeap()源代码

```
static bool addNewHeap(HeapSource * hs)
{
    Heap heap;                                          //声明堆

    assert(hs != NULL);
    if (hs->numHeaps >= HEAP_SOURCE_MAX_HEAP_COUNT) {   //如果堆的数量过大
        LOGE("Attempt to create too many heaps (%zd >= %zd)",
                hs->numHeaps,HEAP_SOURCE_MAX_HEAP_COUNT);
        dvmAbort();
        return false;                                   //返回错误
    }

    memset(&heap,0,sizeof(heap));                       //初始化堆空间

    void * sbrk0= contiguous_mspace_sbrk0(hs->heaps[0].msp);
    char * base= (char * )ALIGN_UP_TO_PAGE_SIZE(sbrk0);
    size_t overhead= base - hs->heaps[0].base;
    assert(((size_t)hs->heaps[0].base & (SYSTEM_PAGE_SIZE - 1))==0);

    if (overhead+ HEAP_MIN_FREE >= hs->maximumSize) {
        LOGE_HEAP("No room to create any more heaps "   //打印错误信息
                 "(%zd overhead,%zd max)",
                 overhead,hs->maximumSize);
        return false;
    }

    heap.maximumSize= hs->growthLimit - overhead;       //设置堆的各个成员变量
    heap.concurrentStartBytes= HEAP_MIN_FREE - CONCURRENT_START;
    heap.base= base;
    heap.limit= heap.base+ heap.maximumSize;
    heap.msp= createMspace(base,HEAP_MIN_FREE,hs->maximumSize - overhead);
    if (heap.msp== NULL) {                              //堆空间为空,返回错误
        return false;
    }

     /** 不允许 soon-to-be-old 堆生长 * /
    hs->heaps[0].maximumSize= overhead;                 //设置成员变量
    hs->heaps[0].limit= base;
    mspace msp= hs->heaps[0].msp;
    mspace_set_max_allowed_footprint(msp,mspace_footprint(msp));
     /** 把新堆放在列表中,在 heaps[0]。向下移动已经存在的堆 * /
    memmove(&hs->heaps[1],&hs->heaps[0],hs->numHeaps * sizeof(hs->heaps[0]));
    hs->heaps[0]= heap;
    hs->numHeaps++;
```

```
    return true;                                    //返回真
}
```

addNewHeap 函数是 HeapSource.cpp 文件中的关键函数，其主要功能是添加额外的堆到堆资源中。如果有太多的堆或者没有充足的空间来添加其他的堆时，返回 false。首先获取入口参数 hs，然后判断 hs→numHeaps ＞＝HEAP_SOURCE_MAX_HEAP_COUNT 是否成立，"HEAP_SOURCE_MAX_HEAP_COUNT"代表堆的最大数量。若成立则发出信息"尝试创建过多的堆"，停止虚拟机，返回 false 结束；若不成立，则对结构体和数组进行清零，初始化新申请的内存。使堆字符与页大小对齐，定义堆数组的底部为 overhead。接着判断 overhead＋HEAP_MIN_FREE ＞＝hs→maximumSize 是否成立，若成立则发出信息"没有更多的空间创建堆"，返回 false 结束；若不成立，则对堆进行初始化，接着判断 heap.msp＝＝NULL 是否成立。若成立则返回 false 结束；若不成立，则对堆内存空间进行初始化，设置内存空间为从系统得到的最大字节数。然后进行块拷贝，把新堆放在列表中，返回 true，结束。

4. allocMarkStack 函数

allocMarkStack 在 dalvik/vm/alloc/HeapSource.cpp 中定义，源代码如下所示，函数流程图如图 2.6 所示。

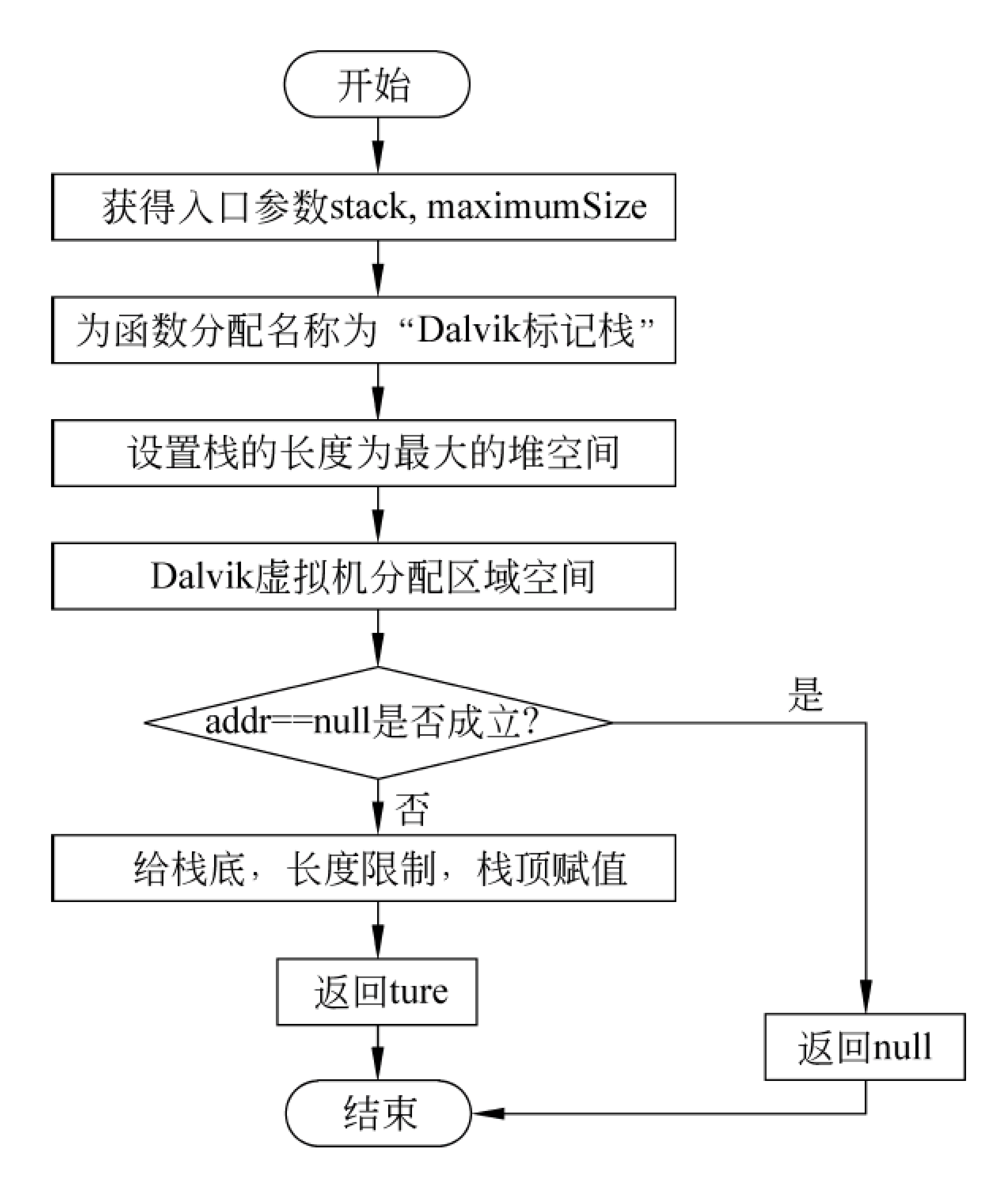


图 2.6　allocMarkStack 函数流程图

代码清单 2.6　dalvik/vm/alloc/HeapSource.cpp：allocMarkStack()源代码

```
static bool allocMarkStack(GcMarkStack * stack,size_t maximumSize)
{
    const char * name="dalvik-mark-stack";
```

```
    void * addr;
    assert(stack != NULL);
    stack->length=maximumSize *  sizeof(Object* ) /
        (sizeof(Object)+HEAP_SOURCE_CHUNK_OVERHEAD);
    addr=dvmAllocRegion(stack->length,PROT_READ | PROT_WRITE,name);
    if (addr==NULL) {                                //如果地址为空,返回假
        return false;
    }
    stack->base=(const Object * *)addr;              //设置各个成员变量
    stack->limit=(const Object * *)((char * )addr+stack->length);
    stack->top=NULL;
    madvise(stack->base,stack->length,MADV_DONTNEED);
    return true;
}
```

allocMarkStack 函数是分配内存空间的关键函数，首先获取 stack，maximumSize 两个入口参数，然后定义一个常量字符指针 * name="dalvik-mark-stack"和空指针 void * addr。判断栈是否为空。栈不为空时，设置栈长度为对象最大分配空间大小 stack→length = maximumSize。addr 赋值为 Dalvik 虚拟机分配区域地址，其参数包括(stack→length，PROT_READ | PROT_WRITE，name)。然后对 addr 进行判断，若 addr==NULL 成立则返回 false，若不成立则继续执行。接着为栈定义 stack→base=(const Object * *)addr，栈底为对象地址；stack→limit=(const Object * *)((char *)addr+stack→length)，栈的长度限制为对象地址+栈的长度；stack→top=NULL，栈顶为空。最后返回 true。

5. dvmHeapSourceAlloc 函数

dvmHeapSourceAlloc 在 dalvik/vm/alloc/HeapSource. cpp 中定义，源代码如下所示，函数流程图如图 2.7 所示。

代码清单 2.7 dalvik/vm/alloc/HeapSource. cpp：dvmHeapSourceAlloc()源代码

```
void*  dvmHeapSourceAlloc(size_t n)
{
    HS_BOILERPLATE();

    HeapSource * hs=gHs;                             //声明堆资源
    Heap*  heap=hs2heap(hs);
    if (heap->bytesAllocated+n >hs->softLimit) {     //如果超过限制,返回空
        LOGV_HEAP("softLimit of %zd.%03zdMB hit for %zd-byte allocation",
                  FRACTIONAL_MB(hs->softLimit),n);
        return NULL;
    }
    void*  ptr=mspace_calloc(heap->msp,1,n);         //分配空间
    if (ptr==NULL) {                                 //分配结果为空,返回空
        return NULL;
    }
```

```
        countAllocation(heap,ptr);
        if (gDvm.gcHeap->gcRunning || !hs->hasGcThread) {
            return ptr;
        }
        if (heap->bytesAllocated > heap->concurrentStartBytes) {
            dvmSignalCond(&gHs->gcThreadCond);
        }
        return ptr;
}
```

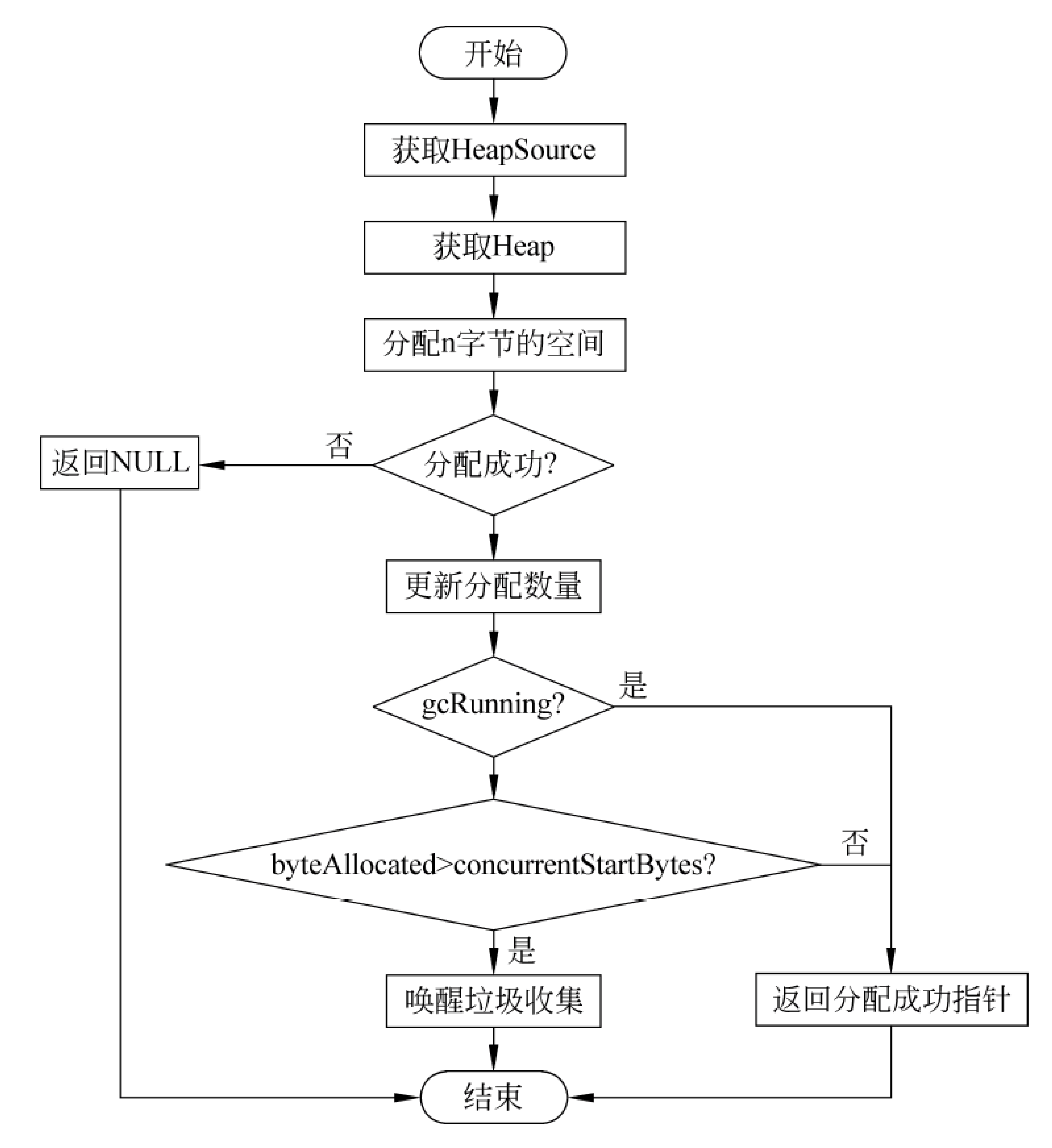


图 2.7　dvmHeapSourceAlloc 函数流程图

dvmHeapSourceAlloc 函数是 Dalvik 虚拟机内存分配部分负责分配内存空间的关键函数，分配过程是在 HeapSource 上的底层内存资源 Heap 上进行的。首先获取虚拟机系统的 HeapSource 资源和 Heap 资源，然后判断分配空间是否超出空间的限制，即 heap→bytesAllocat ed＋n＞hs→softLimit 的真值是否为真，如果为真，证明当前空间的最大值不足以分配指定大小的空间，返回 NULL 指针并打印信息。否则如果为假，则继续执行分配过程，调用底层函数 mspace_calloc 在 Heap 的内存区域上分配一个 n 字节大小的空间，返回一个指向内存空间的指针，如果指针为空，则内存分配失败，返回一个 NULL 指针，若不为空，则调用 countAllocation 函数更新分配结果，将分配的空间大小和分配的对象数量进行更新，即将 bytesAllocated 和 objectsAllocated 数据进行更新。然后判断垃圾回收过程是否正在进行，即 gcRunning 是否为真，如果为真，则说明垃圾回收过程正在执行，返回内存

分配指针，分配过程成功完成。否则为假，则继续向下执行。判断 bytesAllocated 是否大于 concurrentStartBytes，即已经分配的比特大小是否大于并行垃圾回收启动的内存使用量，如果是，则说明分配 n 比特的内存空间之后内存空间使用量过大，需要唤醒垃圾收集器进行垃圾回收，回收完成后返回指向内存区域的指针。若没有超过，则直接返回。

这个函数应用于 Dalvik 虚拟机内存分配的 4 次内存分配尝试中，每一次内存分配都将调用此函数，此函数是一个接触底层分配的一个函数，操作的内存区域都是实实在在的底层内存区。

2.2.3 内存分配流程

虚拟机内存分配的底层依赖是基于 Doug Lea 编写的 dlmalloc 内存分配器，在 Heap 上完成。内存分配过程伪代码如下所示。

代码清单 2.8 内存分配过程伪代码

```
Object * dvmAllocObject(ClassObject * clazz,int flags) {
    n=get object size form class object clazz
    first try: allocate n bytes from heap                                //第一次尝试
    if first try failed {                                                //如果第一次尝试失败
        run garbage collector without collecting soft references         //第一次垃圾收集
        second try: allocate n bytes from heap                           //第二次尝试
    }
    if second try failed {                                               //如果第二次尝试失败
        third try: grow the heap and allocate n bytes from heap          //第三次尝试
    }
    if third try failed {                                                //如果第三次尝试失败
        run garbage collector with collecting soft references            //第二次垃圾收集
    fourth try: grow the hap and allocate n bytes from heap              //第四次尝试
    }
    if fourth try failed,return null pointer,dalvik vm will abort        //如果第四次尝试失败
}
```

可以看出，为了分配内存，虚拟机尽了最大的努力，做了 4 次尝试。如果第一次内存分配失败了，那么进行一次垃圾收集，这次垃圾收集并不收集软引用对象，收集完成之后，再次尝试进行内存分配，如果第二次尝试再次失败，就增长堆的大小，并进行第三次的分配尝试，因为堆是可以在堆生长限制之内进行生长的。如果第三次尝试再次失败，启动垃圾收集器进行垃圾收集，此次收集过程收集软引用对象，收集过程完成之后进行第四次的内存分配尝试，如果分配成功则返回一个指向内存区域的指针，否则，返回空指针并且抛出异常，虚拟机暂停工作。在整个内存分配尝试中，进行了两次垃圾收集，第一次不收集 SoftReference，第二次收集 SoftReference。垃圾收集的目的就是及时回收那些不再使用的对象的空间，保证内存堆中有足够的空间可以分配对象。

点拨 SoftReference 是默认引用实现，它会尽可能长时间地存活于虚拟机内，当没有任何对象指向它时 GC 执行后将会被回收；WeakReference，顾名思义，是一个弱引用，当所引用的对象在虚拟机内不再有强引用时，GC 后 WeakReference 将会被自动回收；

SoftReference 与 WeakReference 的特性基本一致，最大的区别在于 SoftReference 会尽可能长地保留引用直到虚拟机内存不足时才会被回收，这一特性使得 SoftReference 非常适合缓存应用。

具体的内存分配流程如图 2.8 所示。

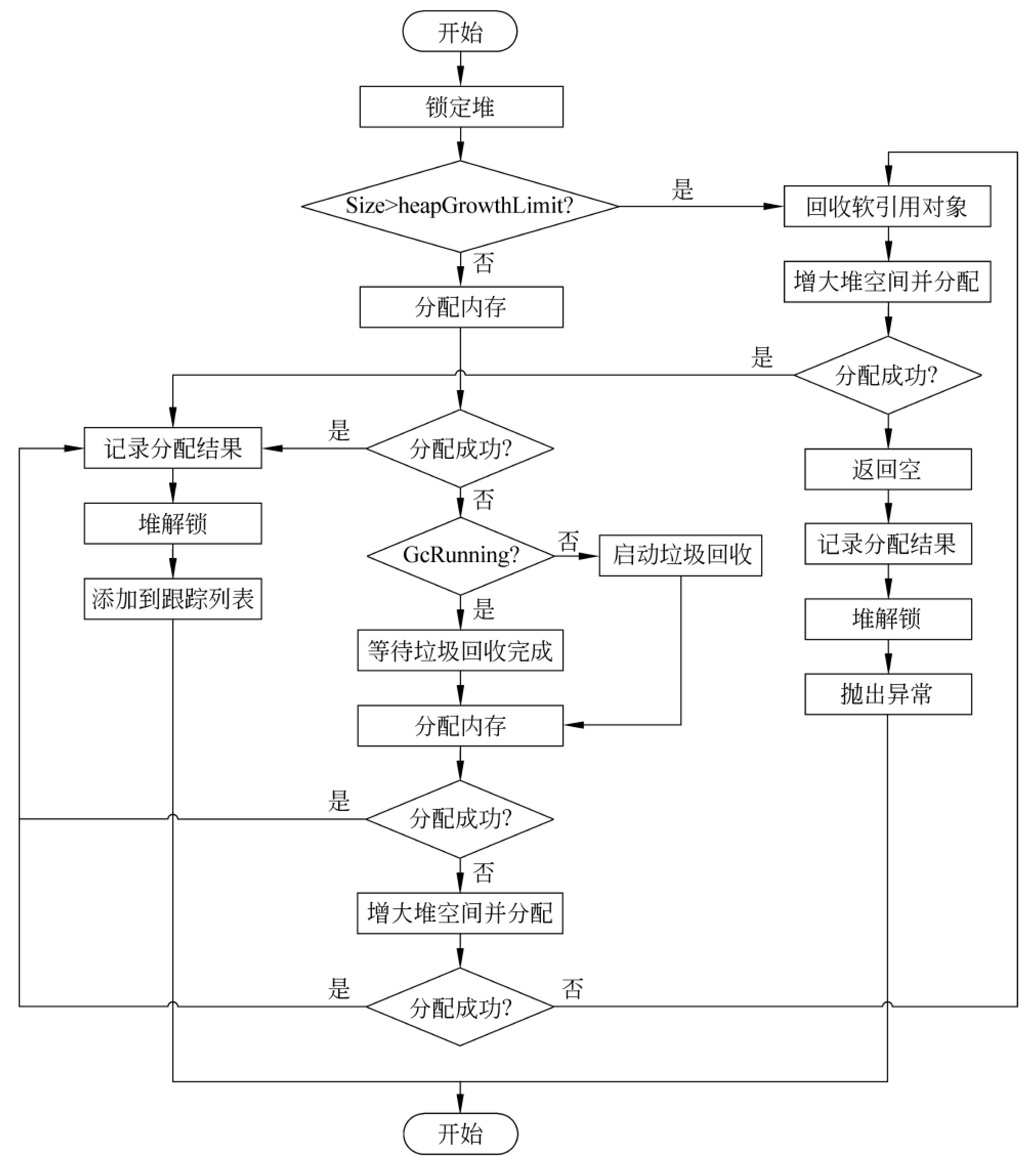


图 2.8　内存分配

第一次垃圾回收时调用函数 gcForMalloc(false)，不回收软引用对象。而第二次垃圾回收时调用函数 gcForMalloc(true)，回收软引用对象。总共进行 4 次分配尝试，若分配成功，返回指向内存区域的指针，否则返回空，并抛出 OutOfMemoryError 异常。以上是内存分配过程的分析结果，结合调试工具可以验证分析结果的正确性。

利用 GDB 自动调试工具可以获取内存管理的堆栈信息，取得堆栈信息后，提取出堆栈中的函数和函数调用关系，利用 Python 脚本转化为 Graphviz 可以识别的 dot 语言，然后使用 dot 命令生成相应的函数流程图，绘制的流程图如图 2.9 所示。

从流程可以看出，内存分配时，dvmMalloc 函数首先调用 dvmLockHeap 函数进行锁定

堆，然后调用 tryMalloc 函数进行分配尝试，其中 tryMalloc 函数中调用了 dvmHeapSourceAlloc 函数进行分配堆资源，分配成功后 dvmMalloc 再调用 dvmUnlockHeap 进行解锁堆，最后调用 dvmAddTrackAlloc 函数进行添加跟踪。可以看出函数执行调用流程与解析代码得到的流程图一致。

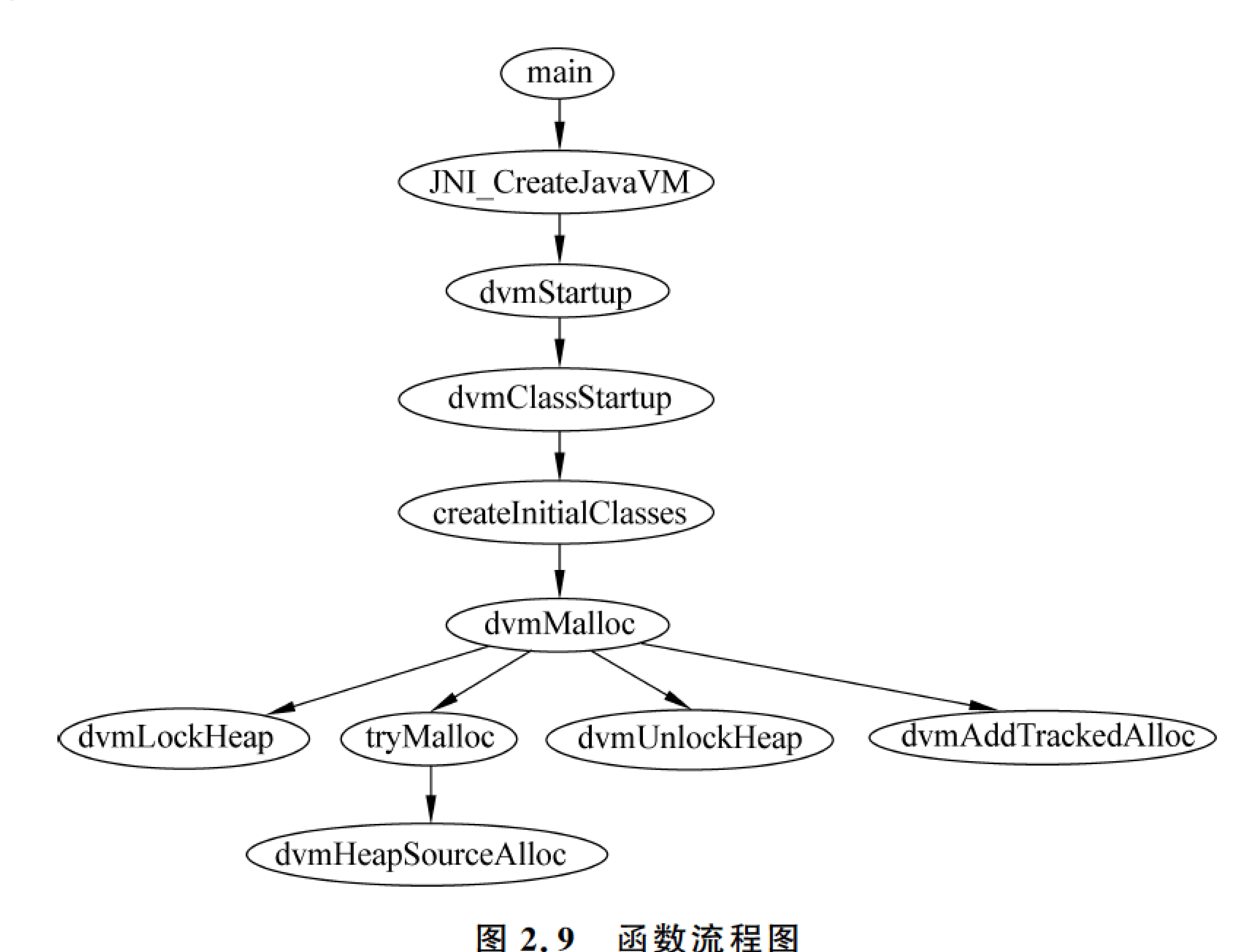


图 2.9　函数流程图

2.3　垃圾回收过程分析

2.3.1　垃圾收集算法

任何垃圾收集算法都必须做两件事，首先它必须检测出垃圾对象。其次，它必须回收垃圾对象所使用的堆空间并还给程序。

垃圾检测通常通过建立一个根对象的集合以及建立一个从这些根对象开始能够触及的对象集合来实现。如果正在执行的程序可以访问到的根对象和某个对象之间存在引用路径，这个对象就是可触及的。对于程序来说，根对象总是可以访问的。从这些根对象开始，任何可以被触及的对象都被认为是"活动"的对象。无法被触及的对象被认为是垃圾，因为它们不再影响程序的未来执行。

虚拟机的根对象集合根据实现不同而不同，包含局部变量中的对象引用和栈帧的操作数栈(以及类变量中的对象引用)、被加载的类的常量池中的对象引用(比如字符串)、传递到本地方法中的没有被本地方法释放的对象引用。任何被根对象引用的对象都是可触及的，从而是活动的，任何被活动的对象引用的对象都是可触及的，程序可以访问任何可触及的对象，所以这些对象应该留在堆中，而对于那些不可触及的对象，程序没有办法访问它们，应该被收集和释放。

区分活动对象和垃圾的两个基本方法是引用计数和跟踪。引用计数垃圾收集器通过为堆中的每一个对象保存一个计数来区分活动对象和垃圾对象。这个计数记录下了对那个对

象的引用次数。跟踪垃圾收集器实际上追踪从根节点开始的引用图。在追踪中遇上的对象以某种方式打上标记，当追踪结束时，没有被打上标记的对象就被判定是不可触及的，可以被当作垃圾收集。

1. 引用垃圾收集器

引用计数是垃圾收集的早期策略。在这种方法中，堆中每一个对象都有一个引用计数。当一个对象被创建了，并且指向该对象的引用被分配给了一个变量，这个对象的引用计数被置为 1。当任何其他变量被赋值为对这个对象的引用时，计数加 1。当一个对象的引用超过了生存期或者被放置一个新的值时，对象的引用计数减 1。任何引用计数为 0 的对象可以被当作垃圾收集。当一个对象被垃圾收集的时候，它引用的任何对象计数值减 1。在这种方法中，一个对象被垃圾收集后可能导致后续其他对象的垃圾收集行动。

这种方法的好处是，引用计数垃圾收集器可以很快地执行，交织在程序的运行之中。这个特性对于程序不能被长时间打断的实时环境很有利。坏处就是，引用计数无法检测出循环(即两个或者更多的对象互相引用)。循环的例子如，父对象有一个对于对象的引用，子对象又反过来引用父对象。这些对象永远都不能计数为 0，就算它们已经无法被执行程序的根对象可触及。还有一个坏处就是，每次引用计数的增加或者减少都带来额外开销。

因为引用计数方法固有的缺陷，这种技术现在已经不为人所接受。现实生活中所遇到的虚拟机更有可能在垃圾收集堆中使用追踪算法。

2. 跟踪收集器

跟踪收集器追踪从根节点开始的对象引用图。在追踪过程中遇到的对象以某种方式打上标记。总的来说，要么在对象本身设置标记，要么用一个独立的位图来设置标记。当追踪结束时，未被标记的对象就知道是无法触及的，从而可以被收集。

基本的追踪算法被称作“标记清除算法”。这个名字指出垃圾收集过程的两个阶段。在标记阶段，垃圾收集器遍历引用树，标记每一个遇到的对象。在清除阶段，未被标记的对象被释放，使用的内存被返回到正在执行的程序。在 Java 虚拟机中，清除步骤必须包括对象的终结。

3. 压缩收集器

Java 虚拟机的垃圾收集器可能有对付堆碎块的策略。标记并清除收集器通常使用的两种策略是压缩和拷贝。这两种方法都是快速地移动对象来减少堆碎块。压缩收集器把活动的对象越过空闲区滑动到堆的一端，在这个过程中，堆的另一端出现一个大的连续空闲区。所有被移动的对象的引用也被更新，指向新的位置。

更新被移动的对象的引用有时候通过一个间接对象引用层可以变得更简单。不直接引用堆中的对象，对象的引用实际上指向一个对象句柄表。对象句柄才指向堆中对象的实际位置。当对象被移动了，只有这个句柄需要被更新为新位置。所有的程序中对这个对象的引用仍然指向这个具有新值的句柄，而句柄本身没有移动。这种方法简化了消除堆碎块的工作，但是每一次对象访问都带来了性能损失。

4. 拷贝收集器

拷贝垃圾收集器把所有的活动对象移动到一个新的区域。在拷贝的过程中它们被紧挨着布置，所以可以消除原本它们在旧区域的空隙。原有的区域被认为都是空闲区。这种方法的好处是对象可以在从根对象开始的遍历过程中随着发现而被拷贝，不再有标记和清除的区分。对象被快速拷贝到新区域，同时转向指针仍然留在原来的位置。转向指针可以让垃圾收集器发现已经被转移的对象的引用。然后垃圾收集器可以把这些引用设置为转向指针的值，所以它们现在指向对象的新位置。

一般的拷贝收集器算法被称为“停止并拷贝”。在这个方案中，堆被分为两个区域，任何时候都只使用其中的一个区域。对象在同一个区域中分配，直到这个区域被耗尽。此时，程序执行被中止，堆被遍历，遍历时遇到的活动对象被拷贝到另一个区域。当停止和拷贝过程结束时，程序恢复执行。内存将从新的堆区域中分配，直到它也被用尽。那时程序将再次中止，遍历堆，活动对象又被拷贝回原来的区域。这种方法带来的代价就是，对于指定大小的堆来说需要两倍大小的内存。因为任何时候都只能使用其中的一半。

2.3.2 关键数据结构

1. GcHeap 结构体

GcHeap 结构体在 dalvik/vm/alloc/HeapInternal.h 文件中定义，以下为结构体的详细分析。

代码清单 2.9 dalvik/vm/alloc/HeapInternal.h：GcHeap 结构体源代码

```
struct GcHeap {
    HeapSource * heapSource;
     /** 递归过程中找到的 java/lang/ref/Reference 子类实例的链接列表。
     * 这些列表在 GC 每次运行时清除和创建 * /
    Object * softReferences;
    Object * weakReferences;
    Object * finalizerReferences;
    Object * phantomReferences;
    Object * clearedReferences;     /**需要入列的引用对象列表 * /
    GcMarkContext markContext;      /**标记步骤当前的状态,只在一个垃圾回收期间有效 * /
    /** 垃圾回收的 card table * /
    u1*  cardTableBase;
    size_t cardTableLength;
    size_t cardTableMaxLength;
    size_t cardTableOffset;
    bool gcRunning;                    /**用来标记垃圾回收是否运行的变量,避免向 GC 重复询问 * /
     /** 调试控制变量值 * /
    int ddmHpifWhen;
    int ddmHpsgWhen;
    int ddmHpsgWhat;
    int ddmNhsgWhen;
    int ddmNhsgWhat;
};
```

2. HeapBitmap 结构体

HeapBitmap 结构体在 dalvik/vm/HeapBitmap.h 文件中定义，以下为结构体的详细分析。

代码清单 2.10　HeapBitmap 结构体源代码

```
struct HeapBitmap {
    /** 位图数据,指向分配好的匿名内存区域 * /
    unsigned long * bits;
    /**使用过的内存大小 * /
    size_t bitsLen;
    /** 分配器分配的真实的内存大小 * /
    size_t allocLen;
    /** 基地址,相当于位图中的第一位 * /
    uintptr_t base;
    /** 最高地址 * /
    uintptr_t max;
};
```

2.3.3　关键函数

1. dvmHeapMarkRootSet 函数

dvmHeapMarkRootSet 函数在 dalvik/vm/alloc/MarkSweep.cpp 中定义，源代码如下所示，函数流程图如图 2.10 所示。

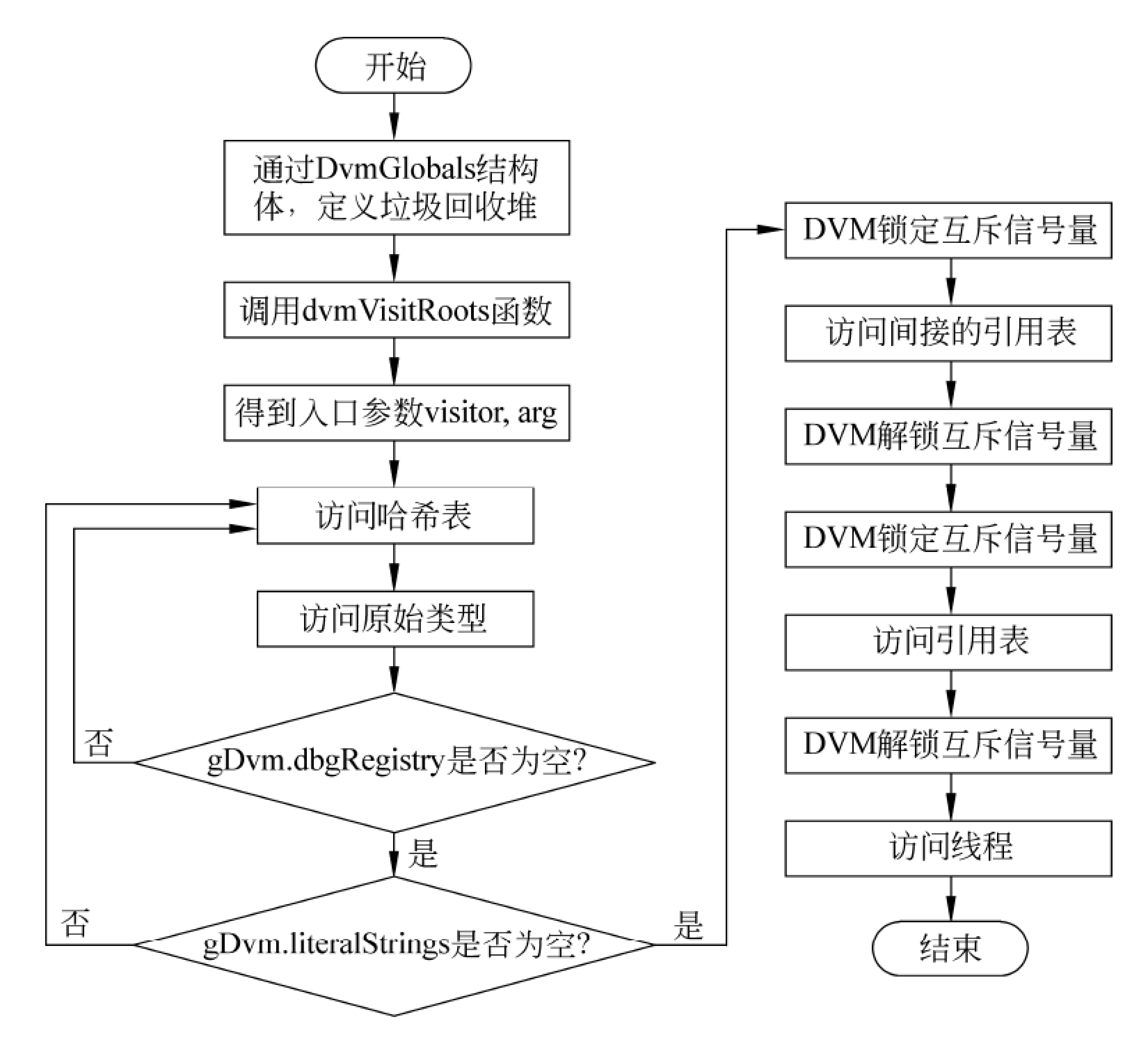


图 2.10　dvmHeapMarkRootSet 函数流程图

代码清单 2.11 dalvik/vm/alloc/MarkSweep.cpp：dvmHeapMarkRootSet()源代码

```
void dvmHeapMarkRootSet()
{
    GcHeap * gcHeap= gDvm.gcHeap;
    dvmMarkImmuneObjects(gcHeap->markContext.immuneLimit);
    dvmVisitRoots(rootMarkObjectVisitor,&gcHeap->markContext);
}
```

dvmHeapMarkRootSet 是对标签进行相关操作中的关键函数，该函数调用了 dvmMarkImmuneObjects 和 dvmVisitRoots 两个函数。该函数主要实现标记一套对象。首先，定义垃圾回收堆。其次，复制存活的 bit 向量内容到标记 bit 向量，使 gcHeap 指向标记内容。然后进行 dvmVisitRoots 访问根操作。再次操作中，首先得到入口参数 visitor 和 arg。接着访问哈希表，其变量为 visitor，gDvm. loadedClasses，ROOT_STICKY_CLASS，arg。访问原始类型，变量为（visitor，arg）。接着判断 gDvm. dbgRegistry 是否为空，若 gDvm. dbgRegistry 不为空，则访问哈希表；若为空则判断 gDvm. literalStrings 是否为空，若 gDvm. literalStrings 不为空，则访问哈希表；若为空则继续执行后续操作 DVM 锁定互斥信号量，访问间接的引用表，DVM 解锁互斥信号量，DVM 锁定互斥信号量，访问引用表，DVM 解锁互斥信号量，访问线程，最终结束。

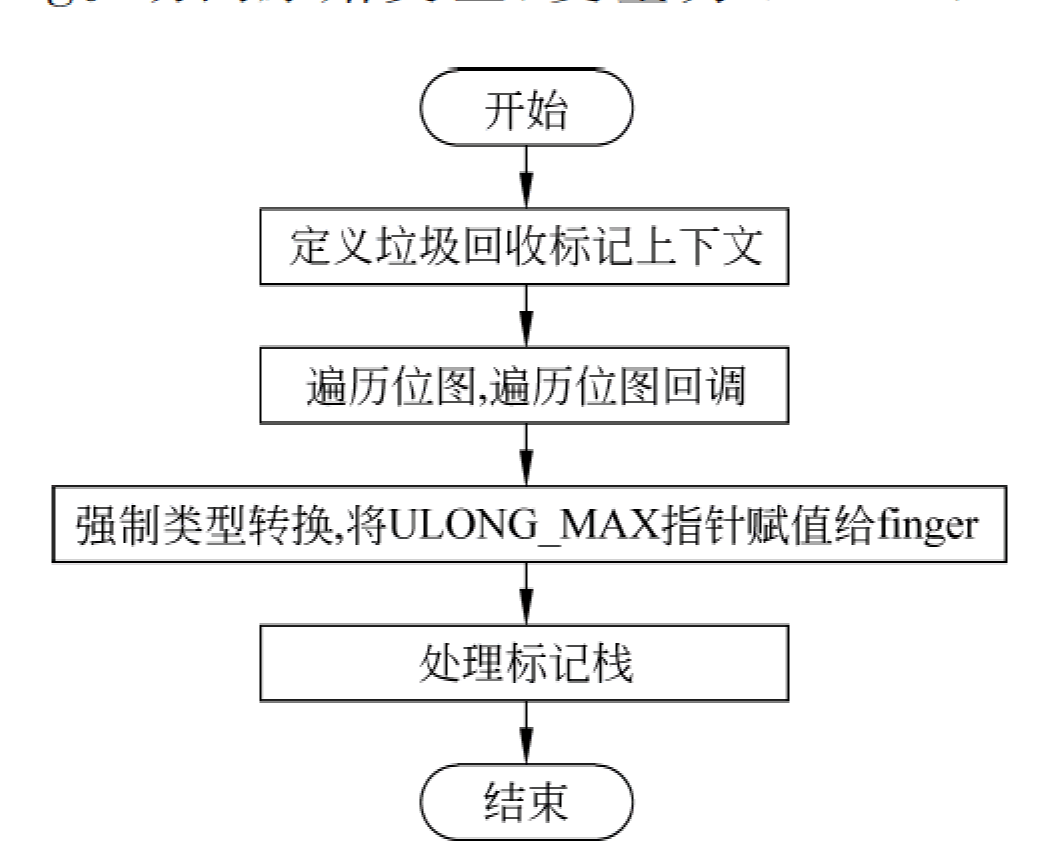


图 2.11 dvmHeapScanMarkedObjects 函数流程图

2. dvmHeapScanMarkedObjects 函数

dvmHeapScanMarkedObjects 在 dalvik/vm/alloc/MarkSweep.cpp 中定义，源代码如下所示，函数流程图如图 2.11 所示。

代码清单 2.12 dalvik/vm/alloc/MarkSweep.cpp：dvmHeapScanMarkedObjects()源代码

```
void dvmHeapScanMarkedObjects(void)
{
    GcMarkContext * ctx= &gDvm.gcHeap->markContext;
    assert(ctx->finger==NULL);
     /** 遍历位图,浏览每个对象 * /
    dvmHeapBitmapScanWalk(ctx->bitmap,scanBitmapCallback,ctx);
    ctx->finger= (void * )ULONG_MAX;
     /** 已经遍历标记位图了。浏览标记栈中留下的所有东西 * /
    processMarkStack(ctx);
}
```

dvmHeapScanMarkedObjects 函数是对标记栈进行相关操作中的关键函数。主要实现给定根标记的位图，找到并且标记所有可触及的对象，函数返回时，整个存活的对象将会被标记并且标记栈为空。首先，将 gDvm. gcHeap→markContext 地址赋给 GcMarkContext

* ctx。遍历位图，遍历位图回调。强制类型转换，将 ULONG_MAX 指针赋值给 finger。处理标记栈，已经遍历标记位图，浏览标记栈留下的东西。

3. processMarkStack 函数

processMarkStack在 dalvik/vm/alloc/MarkSweep.cpp 中定义，源代码如下所示，函数流程图如图 2.12 所示。

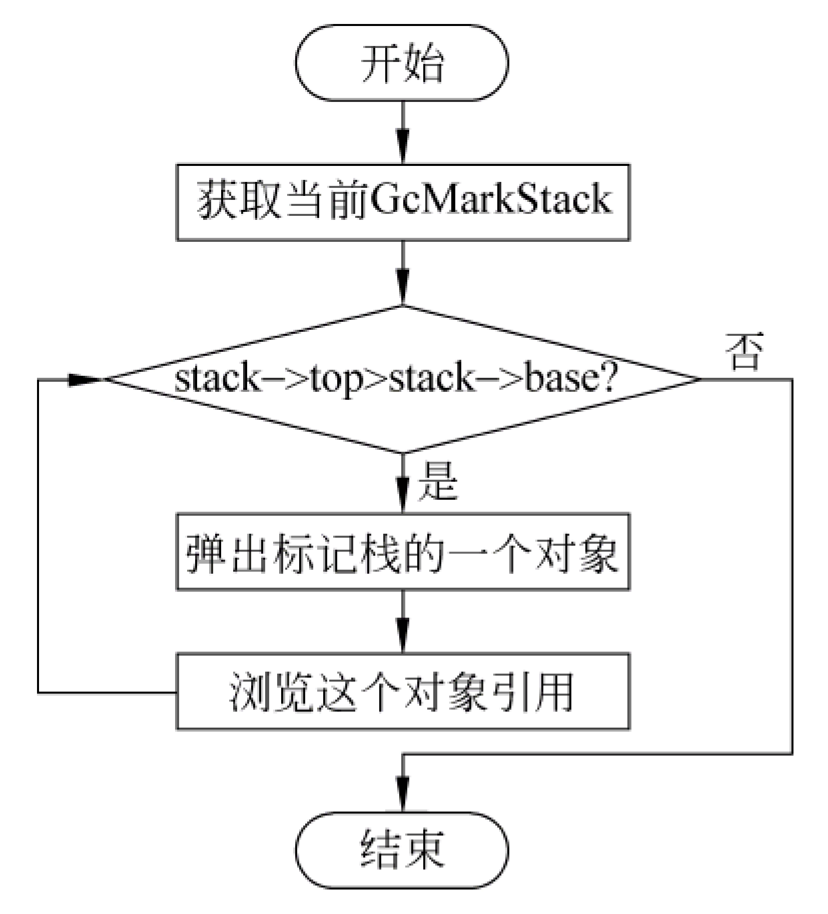


图 2.12　processMarkStack 函数流程图

代码清单 2.13　dalvik/vm/alloc/MarkSweep.cpp：processMarkStack()源代码

```
static void processMarkStack(GcMarkContext * ctx)
{
    assert(ctx != NULL);
    assert(ctx->finger== (void * )ULONG_MAX);
    assert(ctx->stack.top >= ctx->stack.base);
    GcMarkStack * stack= &ctx->stack;
  //声明栈
  while(stack->top > stack->base) {   //如果栈顶大于栈底
        const Object * obj=markStackPop(stack);
        scanObject(obj,ctx);
    }
}
```

processMarkStack 函数是在 Dalvik 虚拟机垃圾回收的标记步骤中用到的处理标记栈的关键函数。我们知道，在标记过程中用标记栈 MarkStack 存储标记的根对象，栈的操作包括弹出 Pop 和压入 Push，在这个函数中为了处理栈中的对象的信息，需要弹出栈顶的对象进行处理。

函数首先获得当前的 GcMarkStack，然后循环处理栈顶的对象，循环条件是 stack→top 大于 stack→base，即栈顶 top 的地址大于栈底 base 的地址，循环体内部执行两个主要操作，一个是弹出栈顶的一个对象，调用函数 markStackPop()，另一个是浏览弹出的这个对象，调用函数 scanObject()。弹出对象时首先栈顶地址减一(--stack→top)，然后返回栈顶的对象元素 * stack→top。浏览对象时，根据浏览的条件来决定浏览对象的类型，浏览类对象 scanClassObject()，浏览数组对象 scanArrayObject()，浏览数据对象 scanDataObject()。

4. ptr2heap 函数

ptr2heap 在 dalvik/vm/alloc/HeapSource.cpp 中定义，源代码如下所示，函数流程图如图 2.13 所示。

代码清单 2.14　dalvik/vm/alloc/HeapSource.cpp：ptr2heap()源代码

```
static Heap * ptr2heap(const HeapSource * hs,const void * ptr)
{
    const size_t numHeaps=hs->numHeaps;   //得到堆的数量
```

```
    if (ptr != NULL) { //如果指针不为空
        for (size_t i = 0; i < numHeaps; i++) {  //每一个堆
            const Heap * const heap = &hs->heaps[i];  //得到堆

            if ((const char * )ptr >= heap->base && (const char * )ptr < heap->limit)
{
                return (Heap * )heap;  //返回堆
            }
        }
    }
    return NULL;        //返回空
}
```

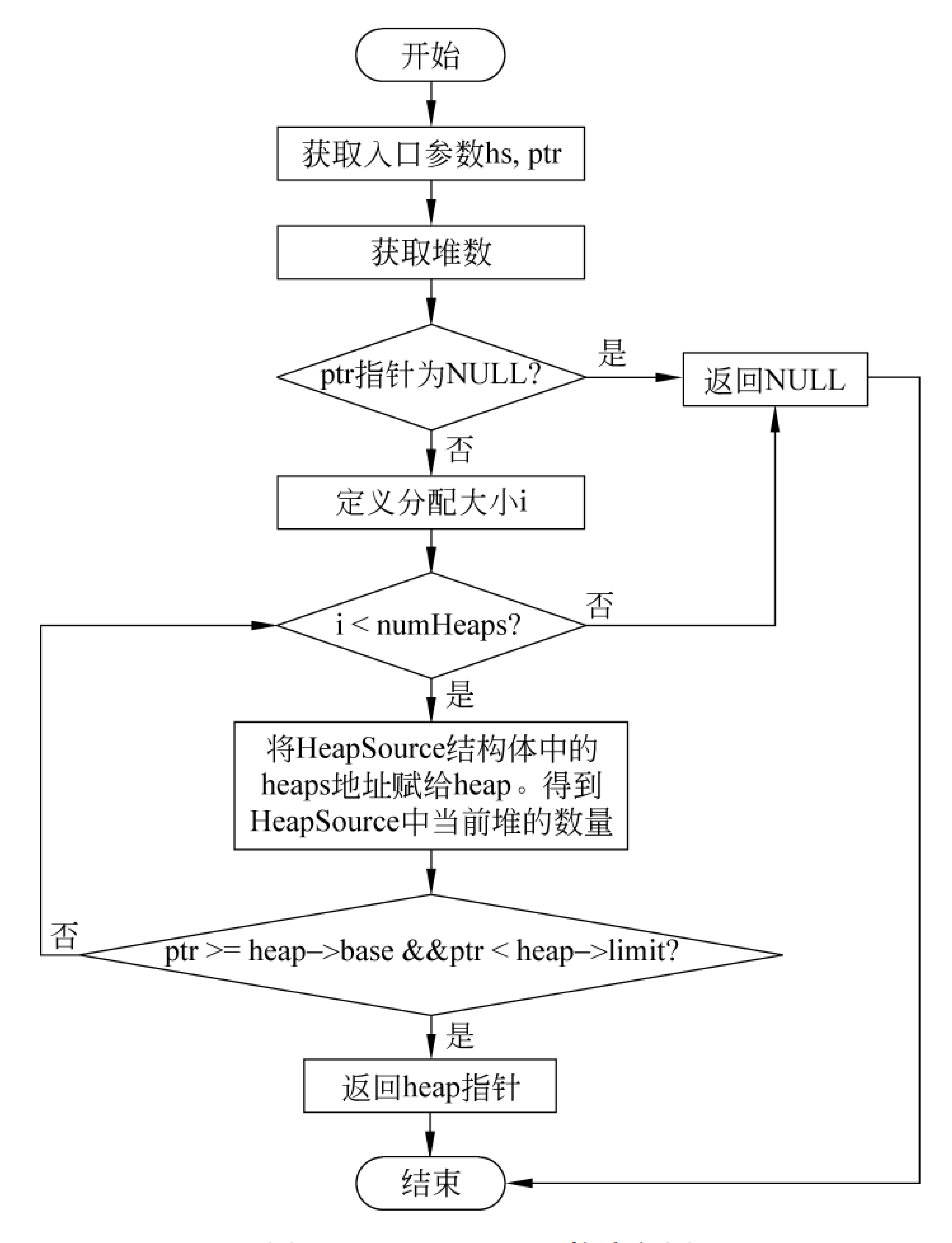


图 2.13 ptr2heap 函数流程图

ptr2heap 主要实现返回<ptr>来自的堆，如果它不来自任何堆则返回空。首先获取入口参数 hs 和 ptr。其次获取堆数。接下来，判断 ptr 指针是否为 NULL，若为 NULL，则返回 NULL；若不为 NULL，则继续执行，定义分配大小 i。判断 i<numHeaps 是否成立，若不成立则返回 NULL，若成立则继续执行，将 HeapSource 结构体中的 heaps 地址赋给 heap。得到 HeapSource 中当前堆的数量。接着，判断(const char *) ptr >= heap→base &&

(const char *)ptr<heap→limit 是否成立，若不成立，则返回上一层判断 i<numHeaps 是否成立并且进行循环；若成立则返回 heap 指针。最终结束。

2.3.4　垃圾回收流程

Dalvik 虚拟机在垃圾回收时使用 Mark Sweep 算法，该算法一般分为 Mark 阶段和 Sweep 阶段。Mark 阶段就是标记出活动对象，使用栈来保存根集合，然后对栈中的每一个元素，递归追踪所有可访问的对象，对于所有可访问的对象，在 markBits 位图中将该对象的内存起始地址对应的位设为 1。这样当栈为空时，markBits 位图就是所有可访问的对象集合。垃圾收集的第二步就是回收内存，在 Mark 阶段通过 markBits 位图可以得到所有可访问的对象集合，而 liveBits 位图表示所有已经分配的对象集合。因此通过比较这两个位图，liveBits 位图和 markBits 位图的差异就是所有可回收的对象集合。

虚拟机的根对象包含局部变量中的对象引用和栈帧的操作数栈(以及类变量中的对象引用)、被加载的类的常量池中的对象引用(比如字符串)、传递到本地方法中的没有被本地方法释放的对象引用。所有这些加入到一个集合中，被称为根集合。然后从根集合开始，递归查找可以从根集合出发访问的对象，Mark 过程又被称为追踪，追踪所有可被访问的对象，任何被根对象引用的对象都是可触及的，从而是活动的，任何被活动的对象引用的对象都是可触及的，程序可以访问任何可触及的对象，所以这些对象应该留在堆中，而对于那些不可触及的对象，程序没有办法访问它们，应该被收集和释放。具体的垃圾回收流程如图 2.14 所示。

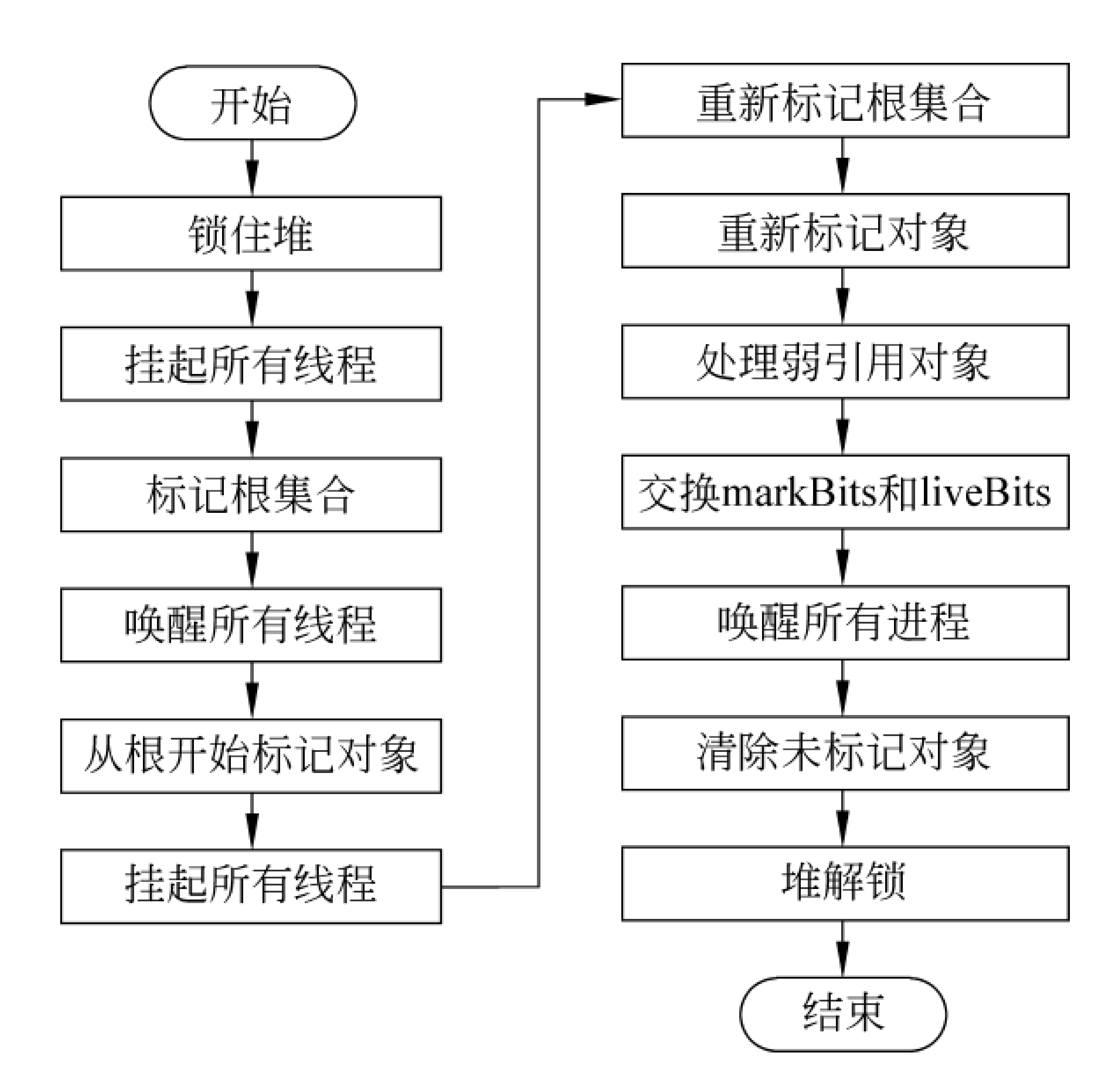


图 2.14　垃圾回收

利用获取到的堆栈信息得到垃圾回收的流程图如图 2.15 所示。从图中可以看到，在垃圾回收时，dvmCollectGarbageInternal 函数依次调用 dvmHeapScanMarkedObjects 函数和 processMarkStack 函数，dvmHeapScanMarkedObjects 函数调用 dvmHeapBitmapScanWalk 函数，并浏览对象。得到的垃圾回收流程与解析代码分析的流程图如图一致。

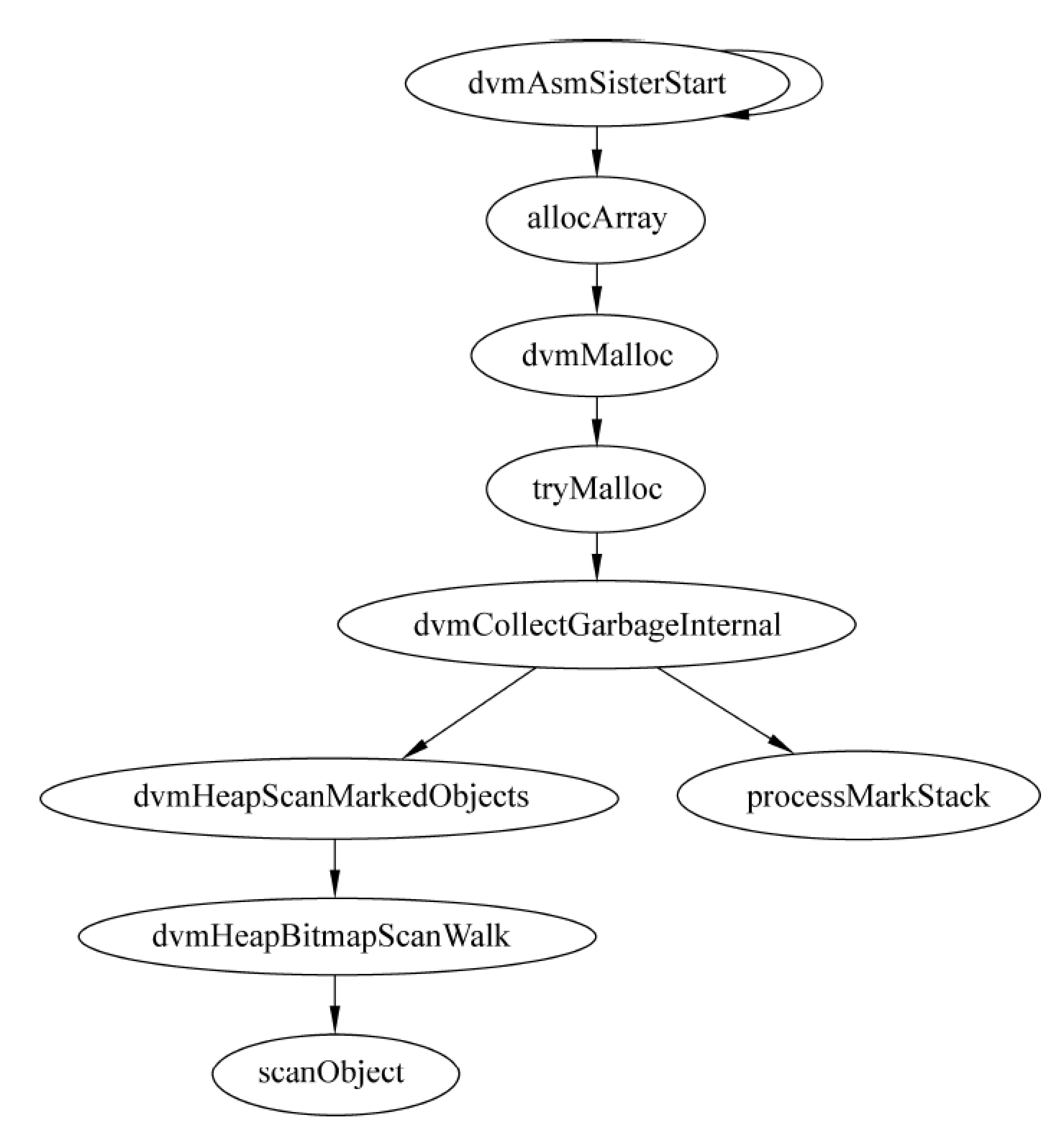


图 2.15 垃圾回收验证流程图

小 结

内存管理是 Android 系统 Dalvik 虚拟机的一个关键部分,优秀可靠的内存管理机制能够为系统及程序的顺畅运行保驾护航。本章在分析源代码的基础上分析了内存管理机制中的关键数据结构以及涉及的关键的函数流程;分析了内存分配算法流程和内存回收的算法原理,较为全面地分析了 Dalvik 虚拟机的内存管理机制。

第3章

JNI模块的原理及实现

本章主要内容

✍ 如何使用 JNI 机制编写应用程序？
✍ Dalvik 虚拟机建立 JNI 机制环境的过程是怎样的？
✍ JNI 机制涉及的关键数据结构有哪些？
✍ JNI 机制涉及的关键函数有哪些？
✍ Java 代码调用 C 代码执行流程是怎样的？
✍ C 代码调用 Java 代码执行流程是怎样的？

大多数的 Android 应用程序都是由 Java 语言编写的，但是在一些情况下本地语言(C/C++)也是不可缺少的，本章将介绍在 Android 应用开发中何时使用本地语言？那么，本地代码和 Java 代码是怎样统一起来的？又是怎样相互调用的呢？本章以分析源代码为基础和根据，向读者一一解答 Dalvik 虚拟机中 JNI 机制的诸多疑问。

3.1 何时使用 JNI

Java 本地调用接口英文全称为 Java Native Interface，简称为 JNI，是 Sun 公司定义的一套编程框架标准接口，允许 Java 代码和本地代码互相调用。本地代码是指那些使用 Java 语言之外的编程语言编写的代码。Android 系统的 Dalvik 虚拟机实现了这套接口，供 Dalvik 虚拟机的 Java 应用与本地代码实现互相调用。

点拨 可以访问 http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html 来获取 Sun 公司针对 JNI 的说明文档。

通常在下列几种情况下使用 JNI 技术。

(1) 需要注重处理速度。和本地代码(C/C++等)相比，Java 代码的执行速度相对慢一些。如果对某段程序的执行速度有较高的要求，建议使用 C/C++ 编写代码。而后在 Java 中通过 JNI 调用基于 C/C++ 编写的部分，常常能够获得更快的运行速度。例如，在图形处理等需要大量计算的情况下通常会广泛地使用 JNI 机制。

(2) 直接进行硬件控制。为了更好地控制硬件，硬件控制代码通常使用 C 语言编写。而后借助 JNI 将其与 Java 层连接起来，从而实现对硬件的控制。Dalvik 虚拟机使用一些本地代码编写的已编译的代码库与硬件、操作系统直接进行交互。

(3) 对既有本地代码的复用。在程序编写过程中,常常会使用一些已经编写好的本地代码(如 C/C++ 代码),既提高了编程效率,又确保了程序的安全性与健壮性。在复用这些本地代码时,就要通过 JNI 本地调用接口来实现。

因此,使用 JNI 的优点主要在于可以直接重用一些数量庞大的本地代码,对于那些性能要求比较高的代码而言,通过 JNI 机制使用本地代码可以增强程序的性能,减少功耗,特别是对于内存较小、CPU 运算速度有限和电池电量不够充沛的嵌入式移动手机设备而言,提高性能减少功耗这一点就显得尤为重要。然而,使用 JNI 调用接口也会带来一些负面影响,比如有些时候失去了平台的可移植性,但是从综合角度考虑,在有些情况下,JNI 机制带来的优点要大于它的缺点。

3.2 JNI 编程示例

3.2.1 加载动态链接库

Android 系统应用层的类都是以 Java 语言编写的,这些 Java 类被编译为包含字节码的 Dex 文件之后,需要依靠 Dalvik 虚拟机来执行。如果 Dalvik 虚拟机在执行 Java 类的过程中,需要和本地代码进行沟通传递信息,这时 Dalvik 虚拟机就需要加载本地代码部分的组件,即由本地代码编译好的.so 动态链接库,然后 Java 类就能够和本地代码中的函数顺利地实现互相调用。可以说,这时 Dalvik 虚拟机扮演着桥梁的角色,让 Java 代码和本地代码实现通过 JNI 机制而相互沟通。Java 代码通过 System.loadLibrary(".so 动态链接库名称")语句加载动态链接库,当虚拟机执行这条语句时,虚拟机就会去 Android 系统的/system/lib 目录下查找指定名称的动态链接库,/system/lib 下存放着 Android 系统的系统应用程序和用户应用程序执行需要的动态链接库,并在解析名称的同时将名称加上"lib"前缀组成完整的动态链接库名称。

3.2.2 声明本地函数

如果在 Java 类中要使用本地函数,那么首先需要在 Java 类中声明它。声明本地函数时,要在函数名前添加"native"关键字进行修饰,例如,如下语句就声明了一个本地函数"add"。在 Java 类中声明完本地函数之后,利用 Java 中的 javac 命令编译 Java 文件生成 class 文件,然后利用 javah 命令使用 class 文件生成 C/C++ 本地语言的.h 头文件,头文件以本地语言的方式声明本地函数,便于后续本地函数实现时使用。

代码清单 3.1 手动编写声明本地函数源代码

```
static native int add(int a,int b);
```

3.2.3 实现本地函数

在 Java 文件中声明完本地函数之后,接下来就应该在 C/C++ 本地语言文件中实现本地函数,将想要实现的函数功能利用本地函数实现。如下即为对上述声明的"fun"本地函数的实现。

代码清单 3.2 手动编写实现本地函数源代码

```
static jint add(JNIEnv * env,jobject thiz,jint a,jint b) {
int result= a+ b;
    LOGI("%d+ %d= %d",a,b,result);
    return result;
}
```

本地函数的参数和返回值类型根据等价约定映射到 Java 语言中的数据类型,Java 语言中存在两种数据类型:基本类型和引用类型。在 Dalvik 虚拟机的 JNI 接口层,也存在着类似的数据类型,与 Java 比较起来,其范围更具严格性。基本数据类型,如 int、float、char 等;引用类型,如类、对象、数组等。如表 3.1 所示为 JNI 基本类型的类型映射。

表 3.1　JNI 基本类型的类型映射

Java 类型	本地类型	描　　述	Java 类型	本地类型	描　　述
boolean	jboolean	C/C++ 8 位整型	int	jint	C/C++ 带符号的 32 位整型
byte	jbyte	C/C++ 带符号的 8 位整型	long	jlong	C/C++ 带符号的 64 位整型
char	jchar	C/C++ 无符号的 16 位整型	float	jfloat	C/C++ 32 位浮点型
short	jshort	C/C++ 带符号的 16 位整型	double	jdouble	C/C++ 64 位浮点型

关于原始类型的类型映射在源代码 jni.h 头文件中声明。在 Java 语言中原始类型可以直接使用,进行赋值、参数传递等操作,而数组、对象等数据类型只能通过引用进行操作和存取,对于引用的类型映射 JNI 也有相应的定义,源代码在头文件 jni.h 中,如表 3.2 所示为 JNI 引用数据类型的类型映射。

表 3.2　JNI 引用数据类型的类型映射

Java 类型	本地类型	描　　述
Object	jobject	任何 Java 对象,或者没有对应 Java 类型的对象
Class	jclass	Class 对象
String	jstring	字符串对象
Object[]	jobjectArray	任何对象的数组
boolean[]	jbooleanArray	布尔型数组
byte[]	jbyteArray	比特型数组
char[]	jcharArray	字符型数组
short[]	jshortArray	短整型数组
int[]	jintArray	整型数组
long[]	jlongArray	长整型数组
float[]	jfloatArray	浮点型数组
double[]	jdoubleArray	双浮点型数组

如图 3.1 所示为 JNI 类型中引用类型的继承关系,可以对具有父子关系的类型进行转换。

从上述的本地函数实现代码中可以看到,一个本地函数有两个固定的函数参数,一个是 JNIEnv * 类型的指针,一个是 jobject 类型,如果本地方法为静态的,那么 jobject 类型的参

数就替换为 jclass 类型的参数。如果除了这两个固定参数还有其他参数，则根据类型映射定义转化为相应类型的参数传入函数中。在本地函数的实现函数体中可以访问 Java 类的成员函数或者成员变量，这时，Java 类中的信息传递到本地函数中。本地函数可以通过 JNI 本地函数调用接口来访问 Java 类中的成员，例如，可以调用 GetMethodID 函数获取 Java 类中方法的 ID，调用 GetFieldID 获取 Java 类中的成员变量 ID，调用 CallVoidMethod 方法调用 Java 类中的方法等。

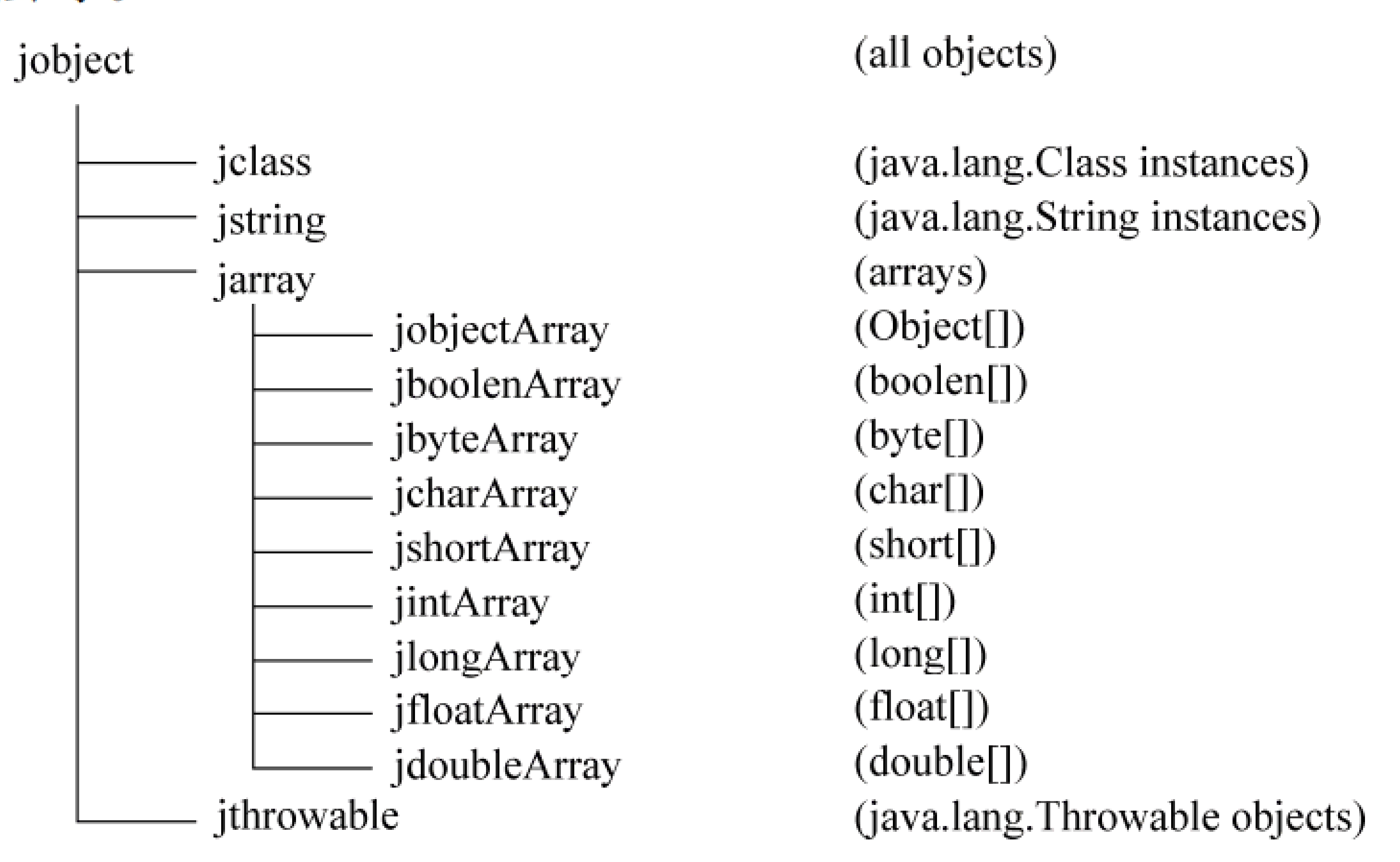


图 3.1 引用类型的继承关系

Dalvik 虚拟机使用了不同于传统 Java JNI 的方式来定义虚拟机的本地函数。其中一个很重要的区别是 Dalvik 虚拟机使用了一种 Java 代码和本地代码函数的映射表数组，并在其中描述了函数的参数和返回值。这个数组的类型为 JNINativeMethod，例如，上述定义的本地函数“fun”利用此类型定义如下。

代码清单 3.3 手动编写 fun 函数的 JNINativeMethod 类型数组定义源代码

```
static JNINativeMethod methods[]= {
    {"fun","(II)I",(void* )fun },
};
```

JNINativeMethod 结构体的定义在头文件 jni. h 中，结构体的具体定义内容如下。

代码清单 3.4 dalvik/libnativehelper/include/nativehelper/jni. h：JNINativeMethod 结构体源代码

```
typedef struct {
    const char*  name;
    const char*  signature;
    void*        fnPtr;
} JNINativeMethod;
```

JNINativeMethod 结构体的成员变量“name”是函数的名字，“signature”是描述函数的参数和返回值的函数签名，“fnPtr”是函数指针，指向本地函数。需要进一步说明的是第二个成员“signature”，签名中的字符实际上是与函数的参数类型和返回类型一一对应的。“()”中的字符表示参数，后面的则代表返回值，例如，“()V”就表示返回值为 void 并且没有参数的一个函数的签名，而本文上述的 fun 方法的签名为“(II)I”，表示返回值为 int，并且具

有两个 int 类型的传入参数。Dalvik 虚拟机的 JNI 机制中对具体的签名字符对应关系作了定义，具体的签名字符对应关系如表 3.3 所示。

表 3.3 和表 3.4 所述都是基本类型，如果 Java 函数的参数是 class，则以“L”开头，以“;”结尾，中间是用“/”隔开的包及类名。而其对应的本地函数名的参数则为 jobject。一个例外是 String 类，其对应的本地类型为 jstring。如果 Java 函数位于一个嵌入类，则用 $ 作为类名间的分隔符。

数组则以“[”开始，用两个字符表示，如表 3.4 所示。

表 3.3　签名字符对应表

字符	Java 类型	本地类型
V	void	void
Z	jboolean	boolean
I	jint	int
J	jlong	long
D	jdouble	double
F	jfloat	float
B	jbyte	byte
C	jchar	char
S	jshort	short

表 3.4　数组签名字符对应表

字符	Java 类型	本地类型
[I	jintArray	int[]
[F	jfloatArray	float[]
[B	jbyteArray	byte[]
[C	jcharArray	char[]
[S	jshortArray	short[]
[D	jdoubleArray	double[]
[J	jlongArray	long[]
[Z	jbooleanArray	boolean[]

3.2.4　实现 JNI_Onload 函数

当 Dalvik 虚拟机执行到 System.loadLibrary()函数时，首先会去执行本地代码库里的 JNI_OnLoad()函数。它的用途有以下两方面。

（1）告诉虚拟机此本地动态链接库使用哪一个 JNI 版本。如果 *.so 库没有提供 JNI_OnLoad()函数，虚拟机会默认该 *.so 挡是使用默认 JNI 版本。由于新版的 JNI 做了许多扩充，如果需要使用 JNI 的新版功能，例如 JNI 1.4 的 java.nio.ByteBuffer，就必须由 JNI_OnLoad()函数来告知虚拟机。

（2）正因为虚拟机执行到 System.loadLibrary()函数时，会立即执行 JNI_OnLoad()函数，所以本地动态链接库的开发者可以通过 JNI_OnLoad()函数来实现本地库内的一些变量的初始化工作，下面就是一个 JNI_OnLoad()函数的具体实现。

代码清单 3.5　手动编写 JNI_Onload()源代码

```
jint JNI_OnLoad(JavaVM* vm,void* reserved)
{
/** 声明变量,并对变量进行初始化 * /
    UnionJNIEnvToVoid uenv;
    uenv.venv=NULL;
    jint result=-1;
    JNIEnv* env=NULL;
/** 获得虚拟机的 env 本地接口函数指针 * /
    if (vm->GetEnv(&uenv.venv,JNI_VERSION_1_4) !=JNI_OK)
    {
```

```
        /** 若发生错误,打印错误信息,进入错误处理代码段 */
            LOGE("ERROR: GetEnv failed");
            goto bail;
        }
        env=uenv.env;
    /** 注册相应的本地函数,这里是“add” */
        if (registerNatives(env) != JNI_TRUE)
        {
        /** 若注册失败,打印错误信息,进入错误处理代码段 */
            LOGE("ERROR: registerNatives failed");
            goto bail;
        }
    /** 设置 JNI 版本为 JNI_VERSION_1_4 */
        result=JNI_VERSION_1_4;
    bail:
        return result;
    }
```

这个 JNI_OnLoad()函数返回 JNI_VERSION_1_4 值给虚拟机,于是 Dalvik 虚拟机知道了其所使用的 JNI 版本。此外,它也做了一些初期的初始化的操作,见以下代码段。

代码清单 3.6 手动编写注册本地函数源代码

```
  if (registerNatives(env) != JNI_TRUE)
{
        LOGE("ERROR: registerNatives failed");
        goto bail;
}
```

将此本地动态链接库提供的各个本地函数注册到 Dalvik 虚拟机里,以便能加快后续调用本地函数的效率。

点拨 在 JNI 编程过程中,JNI_Onload()函数并不是必需的,如果程序中没有实现 JNI_Onload()函数,虚拟机会调用默认的 JNI_Onload()函数。

此外,Dalvik 虚拟机的 JNI 机制还提供 JNI_OnUnload()函数。JNI_OnUnload()函数是与 JNI_OnLoad()相对应的。在加载本地函数库时会立刻调用 JNI_OnLoad()函数来进行本地代码的初期动作;而当虚拟机释放该本地函数库时,则会调用 JNI_OnUnload()函数来进行清除动作。无论虚拟机调用 JNI_OnLoad()函数还是 JNI_Unload()函数,都会将虚拟机的指针传递给它们,其函数原型如下:

```
jint JNI_OnLoad(JavaVM* vm,void* reserved) {    }
jint JNI_OnUnload(JavaVM* vm,void* reserved){    }
```

3.3 JNI 机制环境的建立

在 Zygote 进程启动过程中会创建一个 AndroidRuntime 类的对象,调用 AndroidRuntime::start 方法完成 Zygote 进程的启动工作,并且调用 JNI_CreateJavaVM 函数启动 Dalvik 虚拟机。下面将详细介绍 AndroidRuntime::start 方法和 JNI_

CreateJavaVM 函数。

3.3.1　AndroidRuntime 类的 start 方法

AndroidRuntime::start()方法的主要作用是启动 Android 的运行时环境。在 AndroidRuntime::start()函数中定义 Android root 的目录为"/system"。然后调用 AndroidRuntime::startVm()函数启动虚拟机。启动虚拟机之后调用 AndroidRuntime::startReg()函数注册 Android 系统的内部本地方法。之后声明一个 jobjectArray 类型的数组用来存放将要执行的 Java 类的类名和选项字符串。然后调用 JNI 本地调用接口函数 env→FindClass()查找要执行的类，之后调用 env→GetStaticMethodID()函数取得这个类的 main 方法，然后调用 env→CallStaticVoidMethod()函数执行这个 main 方法。

代码清单 3.7　framework/base/core/jni/AppRuntime.cpp：AndroidRuntime::start()源代码

```
void AndroidRuntime::start(const char* className,const char* options)
{
    LOGD("\n>>>>>> AndroidRuntime START %s<<<<<<\n",
        className != NULL?className : "(unknown)");

    blockSigpipe();
    if (strcmp(options,"start-system-server")==0) {                //验证命令行字符串
        const int LOG_BOOT_PROGRESS_START= 3000;
        LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
                        ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
    }

    const char* rootDir= getenv("ANDROID_ROOT");                   //创建根目录
    if (rootDir==NULL) {
        rootDir= "/system";                                        //创建/system目录
        if (!hasDir("/system")) {                                  //如果/system已存在
            LOG_FATAL("No root directory specified,and /android does not exist.");
            return;
        }
        setenv("ANDROID_ROOT",rootDir,1);
    }

    //const char* kernelHack= getenv("LD_ASSUME_KERNEL");
    //LOGD("Found LD_ASSUME_KERNEL= '%s'\n",kernelHack);
    JNIEnv* env;
    if (startVm(&mJavaVM,&env) != 0) {                             //调用 startVm()
        return;
    }
    onVmCreated(env);
    if (startReg(env)< 0) {                                        //注册函数
        LOGE("Unable to register all android natives\n");
```

```
            return;
        }
        jclass stringClass;
        jobjectArray strArray;
        jstring classNameStr;
        jstring optionsStr;

        stringClass=env->FindClass("java/lang/String");                    //查找类
        assert(stringClass !=NULL);
        strArray=env->NewObjectArray(2,stringClass,NULL);                  /创建新数组
        assert(strArray !=NULL);
        classNameStr=env->NewStringUTF(className);
        assert(classNameStr !=NULL);
        env->SetObjectArrayElement(strArray,0,classNameStr);               //设置数组元素
        optionsStr=env->NewStringUTF(options);
        env->SetObjectArrayElement(strArray,1,optionsStr);                 //设置数组元素
        char* slashClassName=toSlashClassName(className);                  //设置类名
        jclass startClass=env->FindClass(slashClassName);                  //查找类
        if (startClass==NULL) { 如果类名为空
            LOGE("JavaVM unable to locate class '%s'\n",slashClassName);
        } else {
            jmethodID startMeth=env->GetStaticMethodID(startClass,"main",
                "([Ljava/lang/String;)V");                                 //找到 main 方法
            if (startMeth==NULL) {
                LOGE("JavaVM unable to find main() in '%s'\n",className);
            } else {
                env->CallStaticVoidMethod(startClass,startMeth,strArray);          //调用函数
#if 0
                if (env->ExceptionCheck())                                 //检查异常
                    threadExitUncaughtException(env);
#endif
            }
        }
        free(slashClassName);                                              //释放资源
        LOGD("Shutting down VM\n");
        if (mJavaVM->DetachCurrentThread() !=JNI_OK)
          LOGW("Warning: unable to detach main thread\n");
        if (mJavaVM->DestroyJavaVM() !=0)
          LOGW("Warning: VM did not shut down cleanly\n");
    }
```

AndroidRuntime::startVm()函数的主要作用是启动 Dalvik 虚拟机。首先通过调用 property_get()函数来获取 Android 系统的各个属性，Android 系统的属性包括两部分：文件保存的持久属性和每次开机导入的 cache 属性。将获得的系统属性存放在各个相关数组

中，并声明了一个 JavaVMOption 结构体类型的向量 Vector<JavaVMOption>mOption，用来存放一部分获得的系统属性，调用 mOption.add(opt)函数将属性添加到向量中。设置 version 为 JNI_VERSION_1_4，然后调用 JNI_CreateJavaVM()函数建立并初始化 Dalvik 虚拟机。

点拨 Android 系统运行需要的资源也是在 Zygote 进程创建时加载进来的。

AndroidRuntime::startReg()函数的主要作用是注册 Android 系统的内部本地函数。函数调用了 register_jni_procs(gRegJNI, NELEM(gRegJNI), env)进行注册，gRegJNI 参数是一个 RegJNIRec 结构体类型的数组，里面存放的是执行每个需要注册的函数。

3.3.2 JNI_CreateJavaVM()函数

JNI_CreateJavaVM()的主要功能是创建 Dalvik 虚拟机，在 dalvik/vm/Jni.cpp 中定义，当前的线程会变为虚拟机的主线程。首先程序检查 JavaVMInitArgs * 类型参数 args 的 version 成员变量是否大于 JNI_VERSION_1_2，如果小于 JNI_VERSION_1_2，则返回错误信息 JNI_EVERSION。然后建立 JNIEnv 和 VM 的结构，声明一个 JavaVMExt * 类型的变量 pVM，并对其分配内存空间和初始化，并且声明一个 JNIEnvExt * 类型的变量 pEnv，调用 dvmCreateJNIEnv()函数创建 JNI 本地调用接口函数表指针，并返回给变量 pEnv。如图 3.2 所示为 Dalvik 虚拟机创建的流程。如代码清单 3.8 所示为 JNI_CreateJavaVM 的源代码。

代码清单 3.8 dalvik/vm/Jni.cpp：JNI_CreateJavaVM()源代码

```
jint JNI_CreateJavaVM(JavaVM* * p_vm,JNIEnv* * p_env,void* vm_args) {
    const JavaVMInitArgs* args= (JavaVMInitArgs* ) vm_args;          //获取参数
    if (args->version< JNI_VERSION_1_2) {                             //检查 JNI 版本
        return JNI_EVERSION;
    }

    // TODO: don't allow creation of multiple VMs - one per customer for now

    memset(&gDvm,0,sizeof(gDvm));                                     //初始化虚拟机内存区域

    JavaVMExt* pVM= (JavaVMExt* ) malloc(sizeof(JavaVMExt));          //分配变量
    memset(pVM,0,sizeof(JavaVMExt));                                  //初始化变量内存空间
    pVM->funcTable= &gInvokeInterface;                                //设置成员变量的值
    pVM->envList=NULL;
    dvmInitMutex(&pVM->envListLock);                                  //初始化信号量

    UniquePtr< const char* []> argv(new const char* [args->nOptions]);
    memset(argv.get(),0,sizeof(char* ) * (args->nOptions));

      int argc= 0;
    for (int i= 0; i< args->nOptions; i++) {                          //检查每一个参数选项
```

```
    const char* optStr=args->options[i].optionString;
    if (optStr==NULL) {
        dvmFprintf(stderr,"ERROR: CreateJavaVM failed: argument %d was NULL\n",i);
        return JNI_ERR;
    } else if (strcmp(optStr,"vfprintf")==0) {                    //检查参数
        gDvm.vfprintfHook= (int (*)(FILE *,const char*,va_list))args->options[i].extraInfo;
    } else if (strcmp(optStr,"exit")==0) {
        gDvm.exitHook= (void (*)(int)) args->options[i].extraInfo;
    } else if (strcmp(optStr,"abort")==0) {
        gDvm.abortHook= (void (*)(void))args->options[i].extraInfo;
    } else if (strcmp(optStr,"sensitiveThread")==0) {
        gDvm.isSensitiveThreadHook= (bool (*)(void))args->options[i].extraInfo;
    } else if (strcmp(optStr,"-Xcheck:jni")==0) {
        gDvmJni.useCheckJni=true;
    } else if (strncmp(optStr,"-Xjniopts:",10)==0) {
        char* jniOpts=strdup(optStr+10);
        size_t jniOptCount=1;
        for (char* p=jniOpts; *p !=0; ++p) {
            if (*p==',') {
                ++jniOptCount;
                *p=0;
            }
        }
        char* jniOpt=jniOpts;
        for (size_t i=0; i<jniOptCount; ++i) {                    //检查每一个 jni 选项
            if (strcmp(jniOpt,"warnonly")==0) {                   //检查参数
                gDvmJni.warnOnly=true;
            } else if (strcmp(jniOpt,"forcecopy")==0) {
                gDvmJni.forceCopy=true;
            } else if (strcmp(jniOpt,"logThirdPartyJni")==0) {
                gDvmJni.logThirdPartyJni=true;
            } else {
                dvmFprintf(stderr,"ERROR: CreateJavaVM failed: unknown -Xjniopts option '%s'\n",
                        jniOpt);
                return JNI_ERR;
            }
            jniOpt +=strlen(jniOpt)+1;
        }
        free(jniOpts);                                            //释放 jni 选项
    } else {
        /* regular option */
        argv[argc++]=optStr;
    }
}
```

```
    if (gDvmJni.useCheckJni) {                                  //如果使用 checkjni
        dvmUseCheckedJniVm(pVM);                                //调用 dvmUseCheckJniVm()
    }

    if (gDvmJni.jniVm != NULL) {                                //如果 jniVm 不为空
        dvmFprintf(stderr,"ERROR: Dalvik only supports one VM per process\n");
        return JNI_ERR;                                         //返回错误
    }
    gDvmJni.jniVm= (JavaVM* ) pVM;

    JNIEnvExt * pEnv= (JNIEnvExt * ) dvmCreateJNIEnv(NULL);
    gDvm.initializing= true;                                    //初始化成功
    std::string status=
            dvmStartup(argc,argv.get(),args->ignoreUnrecognized,(JNIEnv* )pEnv);
                                                        //调用 dvmStartup 启动本地服务
    gDvm.initializing= false;

    if (!status.empty()) {
        free(pEnv);
        free(pVM);
        LOGW("CreateJavaVM failed: %s",status.c_str());
        return JNI_ERR;
    }
    dvmChangeStatus(NULL,THREAD_NATIVE);                        //改变虚拟机状态
    * p_env= (JNIEnv * ) pEnv;
    * p_vm= (JavaVM* ) pVM;
    LOGV("CreateJavaVM succeeded");
    return JNI_OK;                                              //返回成功
}
```

JNI_CreateJavaVM()中初始化虚拟机的关键部分是 dvmStartup()函数，在 dvmStartup()函数中建立并初始化 Dalvik 虚拟机的各个组件，包括建立线程管理机制、建立垃圾回收机制，建立寄存器映射，建立类加载器等。其中也有很多关于 JNI 机制的组件初始化工作。

dvmInlineNativeStartup()函数给虚拟机的内联方法表 gDvmInlineOpsTable[]分配内存空间。gDvmInlineOpsTable[]内联函数表包括一些 Dalvik 虚拟机要使用的基础的运算函数，比如字符串比较函数、计算绝对值函数、开平方函数、三角函数计算函数等。

dvmNativeStartup()函数的功能是建立共享库的哈希表。创建一个表长为 4 的可扩展哈希表，用来存放本地调用过程中需要用的动态链接库。

dvmInternalNativeStartup()函数初始化一些类名的哈希值，这些类是 Dalvik 虚拟机内部的类，并提供类的方法。首先，函数声明一个指向 DalvikNativeClass 结构体的指针 classPtr 并初始化为 gDvmNativeMethodSet。gDvmNativeMethodSet 是一个 DalvikNativeClass 类型的数组，存放着这些虚拟机内部的类。然后对 gDvmNativeMethodSet 数组中的每一个

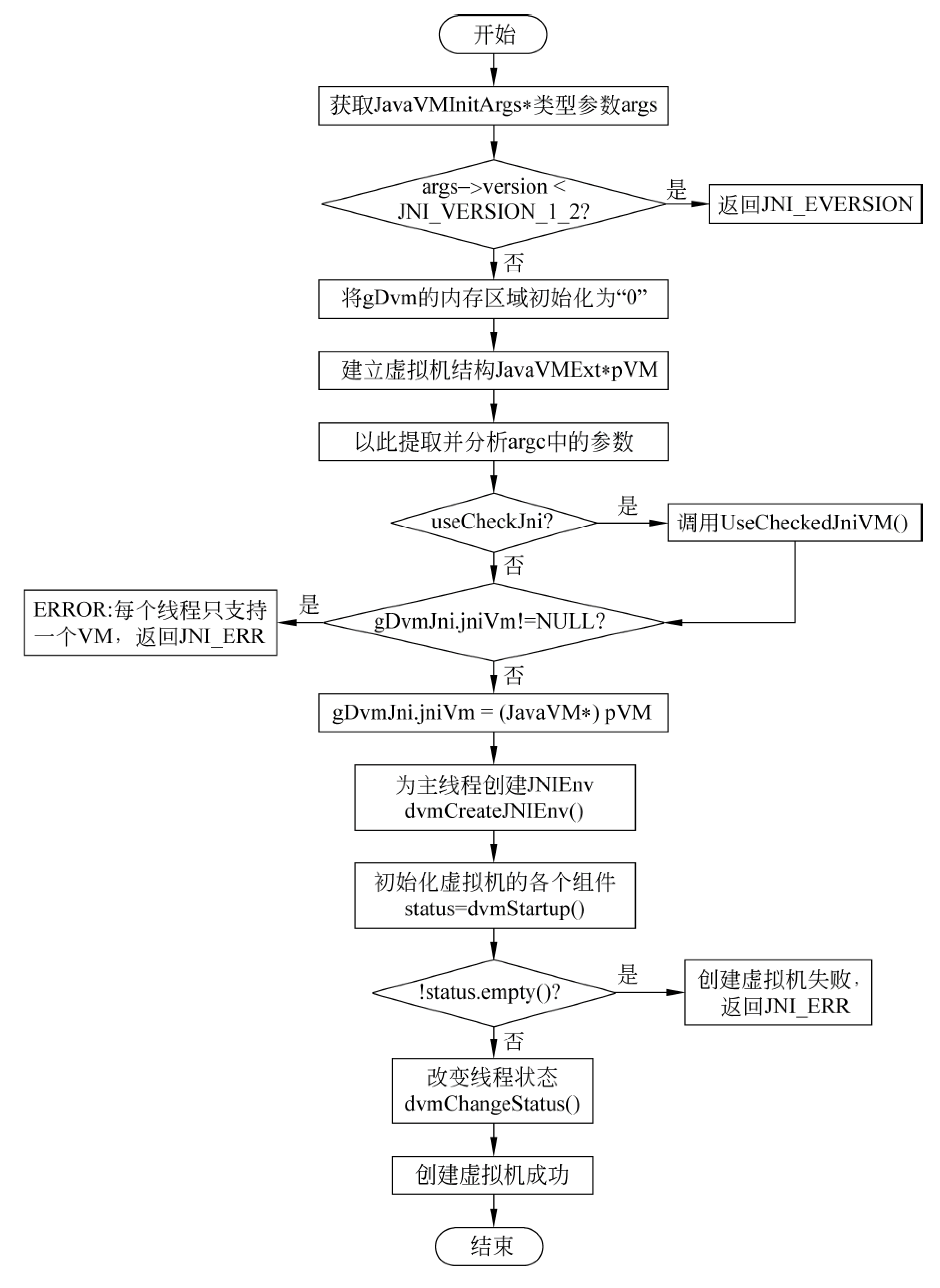


图 3.2 虚拟机创建流程

DalvikNativeClass 类型的元素判断其 classDescriptorHash 成员是否为空，如果不为空，那么就调用 dvmComputeUtf8Hash()函数计算哈希值并赋值给 classDescriptorHash 成员，完成初始化工作。

dvmJniStartup()函数初始化本地调用机制的引用表。函数首先执行 gDvm.jniGlobalRefTable.init(kGlobalRefsTableInitialSize，kGlobalRefsTableMaxSize，kIndirectKindGlobal)，将 Dalvik 虚拟机全局变量 gDvm 的 jniGlobalRefTable 初始化，传入的三个参数均为系统定义的，kGlobalRefsTableInitialSize 是 Global Reference Table 的初始大小，为 512；kGlobalRefsTableMaxSize 为 Global Reference Table 的最大值，为 51 200，这个值是任意定义的，只要小于 64K 即可；kIndirectKindGlobal 是 Indirect reference 种类，定义如下。

```
enum IndirectRefKind {
kIndirectKindInvalid    =0,                   //间接引用非法
kIndirectKindLocal      =1,                   //间接引用是局部引用
kIndirectKindGlobal     =2,                   //间接引用是全局引用
kIndirectKindWeakGlobal=3                     //间接引用是弱全局引用
};
```

这个枚举类型和 jni.h 文件中的 jobjectRefType 是相匹配的。初始化后，就为全局引用表 jniGlobalRefTable 分配了内存空间。然后，执行 gDvm. jniWeakGlobalRefTable. init (kWeakGlobalRefsTableInitialSize，kGlobalRefsTableMaxSize，kIndirectKindWeakGlobal)，原理类似，将 gDvm 的 jniWeakGlobalRefTable 初始化，传入的 kWeakGlobalRefsTableInitialSize 参数为 16，kGlobalRefsTableMaxSize 参数为 51 200，kIndirectKindWeakGlobal 是枚举类型 IndirectRefKind 的一个成员，值为 3。

dvmPreMainForJni()函数完成线程结构体的一些初始化工作，这些初始化工作用以支持本地调用机制。函数首先获得系统的主线程，主线程通常位于线程列表的第一个。然后调用 createFakeEntryFrame()函数在主线程解释栈的顶端创建一个“fake”JNI 帧。栈的顶端在内存的低地址处，栈的底端在内存的高地址处，简单地说就是栈向下生长。然后调用 dvmSetJniEnvThreadId()函数设置 envThreadId，之后调用 dvmSetThreadJNIEnv()函数设置 JNIEnv 域。

registerSystemNatives()函数注册 Android 系统的内部本地方法。首先函数获取到系统的主线程指针，同样位于线程表的第一个。然后执行 self→status = THREAD_NATIVE，将线程的状态切换至 THREAD_NATIVE，这个操作是必需的，必须在允许基于 JNI 的方法注册之前设置此线程状态。然后执行 jniRegisterSystemMethods()函数注册系统本地方法，在 jniRegisterSystemMethods()函数中又分别调用 registerCoreLibrariesJni()函数和 registerJniHelp()函数执行针对于具体类和方法的注册操作。

以上是和本地调用机制直接相关的初始化工作，执行完系统组件的各个初始化工作之后，Dalvik 虚拟机就算建立起来了。

3.4 Java 调用 C 执行流程分析

3.4.1 解释器栈帧结构体

解释器作为 Dalvik 虚拟机的执行引擎在 Dalvik 虚拟机的执行流程中扮演着非常重要的角色，而为了分析函数的执行顺序和流程，必须对解释器栈有一个深刻的理解。如图 3.3 所示为解释器栈的结构简图。

Android 系统的 Dalvik 虚拟机在执行 Android 应用程序时，每一个线程都维护着一个虚拟的解释器方法栈。栈的栈顶在低地址，栈底在高地址，栈是向下生长的。Dalvik 虚拟机的解释器要在执行方法前将栈帧压入解释器栈中。栈帧结构体 Struct StackSaveArea 的定义如表 3.5 所示。

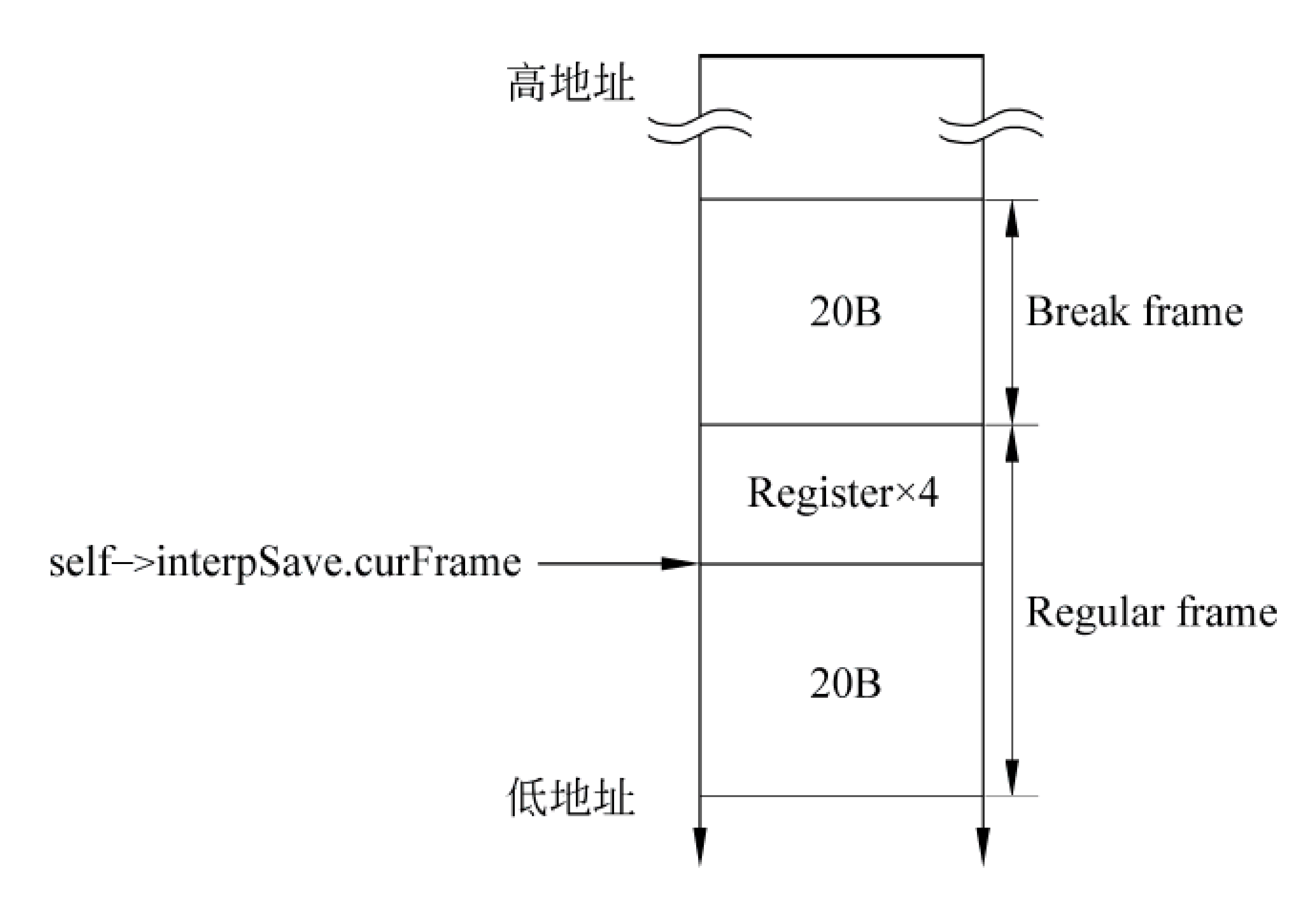


图 3.3 解释器栈简图

表 3.5 栈帧结构体成员变量

成员变量	类　型	变 量 说 明
prevFrame	u4*	保存指向前一个栈帧,如果在栈底,那么为空
savedPc	const u2*	保存的程序计数器
method;	const Method*	保存当前执行的方法指针
localRefCookie	u4	对于 JNI 本地方法:局部引用段的底部
currentPc	const u2*	对于解释方法:保存当前的 PC,为了异常栈追踪和调试跟踪
returnAddr	const u2*	JIT 本地返回指针,如果为 0 是解释方法

代码清单 3.9 dalvik/vm/interp/Stack. h:StackSaveArea 结构体定义

```
struct StackSaveArea {
    u4 *            prevFrame;
    const u2 *        savedPc;
    const Method *     method;
    union {
        u4                  localRefCookie;
        const u2 *        currentPc;
    } xtra;
    const u2 *  returnAddr;
};
```

结构体包括 5 个成员变量,每个成员变量占 4B,整个栈帧占有 20B 的空间。调用一个方法时要压入两个栈帧,其中第一个栈帧叫做 break frame,它的 method 字段为 NULL,break frame 栈帧的作用是处理方法的返回和异常发生的情况。第二个栈帧就是方法本身的栈帧了,可以称之为 regular frame,在 regular frame 栈帧的前部会分配占 registers×4 大小的内存空间,registers 是方法的寄存器数量,一个寄存器是 32 位,占有 4B。这部分内存空间用来保存方法的参数和局部变量。Dalvik 虚拟机解释器可以通过查看栈帧的 method 成员是否为 NULL 来区分这两个栈帧。

点拨 Dalvik 虚拟机解释器在函数执行前调用 dvmPushInterpFrame 函数来进行压栈操作,使栈指针向低地址方向移动,当解释器执行完函数时,调用 dvmPopFrame 函数将栈帧弹出,栈指针向高地址方向移动。

3.4.2　关键函数

1. JNI_CreateJavaVM 函数

JNI_CreateJavaVM(JavaVM * * p_vm，JNIEnv * * p_env，void * vm_args)在 dalvik/vm/Jni.cpp 中定义，源代码如下所示，函数流程图如图 3.4 所示。

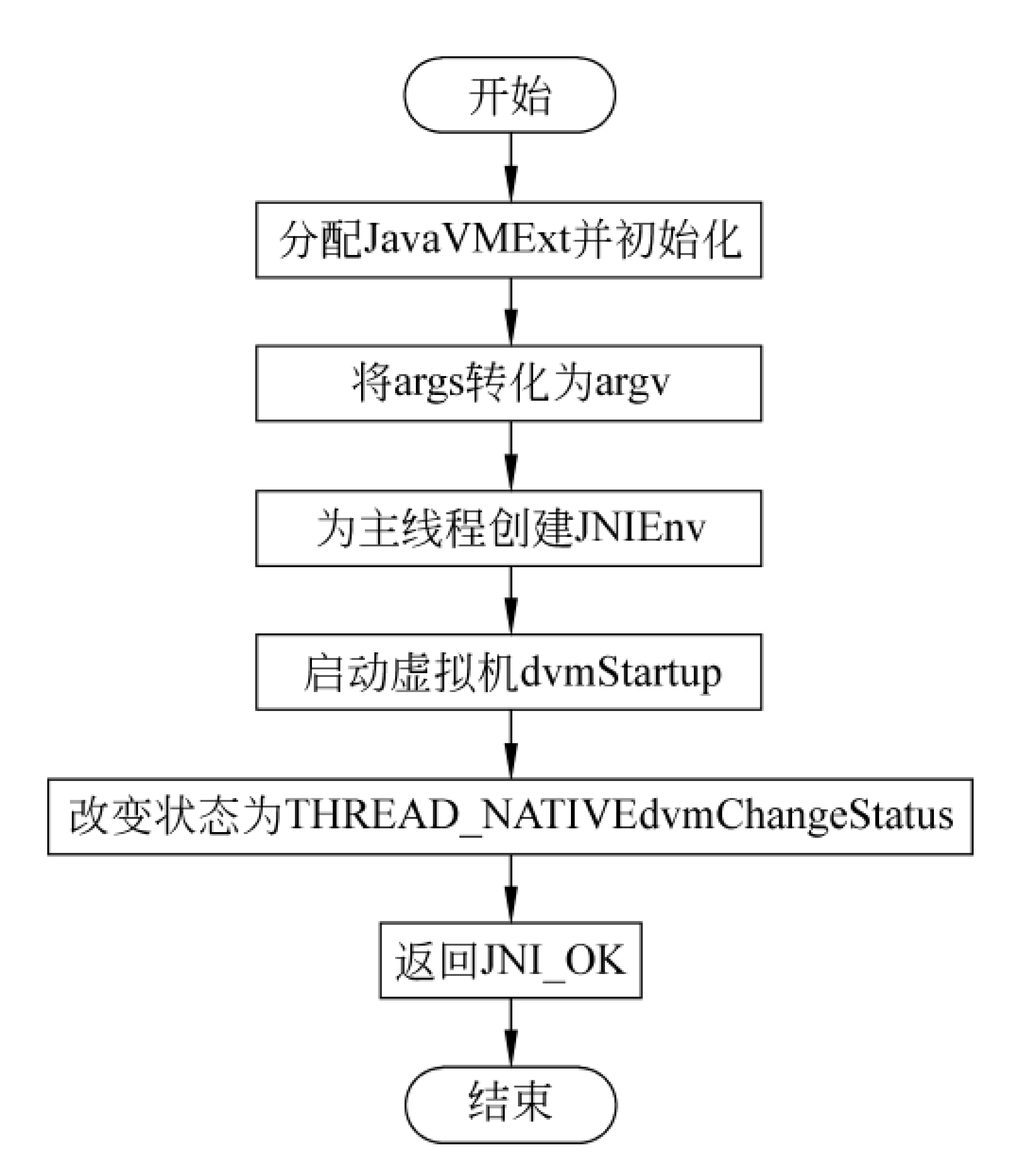


图 3.4　JNI_CreateJavaVM()函数流程图

代码清单 3.10　dalvik/vm/Jni.cpp：JNI_CreateJavaVM()源代码

```
jint JNI_CreateJavaVM(JavaVM* * p_vm,JNIEnv* * p_env,void* vm_args) {
    const JavaVMInitArgs* args= (JavaVMInitArgs* ) vm_args;
/**检查 version是否大于等于 JNI_VERSION_1_2* /
    if (args->version< JNI_VERSION_1_2) {
      return JNI_EVERSION;
    }
/**初始化 gDvm* /
     memset(&gDvm,0,sizeof(gDvm));
     JavaVMExt* pVM= (JavaVMExt* ) malloc(sizeof(JavaVMExt));
/**初始化 JavaVMExt* /
    memset(pVM,0,sizeof(JavaVMExt));
    pVM->funcTable= &gInvokeInterface;
    pVM->envList=NULL;
    dvmInitMutex(&pVM->envListLock);

    UniquePtr< const char* []> argv(new const char* [args->nOptions]);
    memset(argv.get(),0,sizeof(char* ) * (args->nOptions));
/**处理参数中的 nOptions* /
     int argc= 0;
    for (int i=0; i< args->nOptions; i++) {          //检查每一个参数选项
        const char* optStr= args->options[i].optionString;
```

```
        if (optStr==NULL) {
            dvmFprintf(stderr,"ERROR: CreateJavaVM failed: argument %d was NULL\n",i);
            return JNI_ERR;
        } else if (strcmp(optStr,"vfprintf")==0) {
            gDvm.vfprintfHook= (int (* )(FILE * ,const char* ,va_list))args->options[i].extraInfo;
        } else if (strcmp(optStr,"exit")==0) {
            gDvm.exitHook= (void (* )(int)) args->options[i].extraInfo;
        } else if (strcmp(optStr,"abort")==0) {
            gDvm.abortHook= (void (* )(void))args->options[i].extraInfo;
        } else if (strcmp(optStr,"sensitiveThread")==0) {
            gDvm.isSensitiveThreadHook= (bool (* )(void))args->options[i].extraInfo;
        } else if (strcmp(optStr,"-Xcheck:jni")==0) {
            gDvmJni.useCheckJni=true;
        } else if (strncmp(optStr,"-Xjniopts:",10)==0) {
            char* jniOpts=strdup(optStr + 10);
            size_t jniOptCount=1;
            for (char* p=jniOpts; * p !=0; ++p) {
                if (* p==',') {
                    ++jniOptCount;
                    * p=0;
                }
            }
            char* jniOpt=jniOpts;
            for (size_t i=0; i<jniOptCount; ++i) {    //检查每一个参数选项
                if (strcmp(jniOpt,"warnonly")==0) {
                    gDvmJni.warnOnly=true;
                } else if (strcmp(jniOpt,"forcecopy")==0) {
                    gDvmJni.forceCopy=true;
                } else if (strcmp(jniOpt,"logThirdPartyJni")==0) {
                    gDvmJni.logThirdPartyJni=true;
                } else {
                    dvmFprintf(stderr,"ERROR: CreateJavaVM failed: unknown -Xjniopts option '%s'\n",
                            jniOpt);
                        return JNI_ERR;
                    }
                    jniOpt +=strlen(jniOpt) +1;
            }
            free(jniOpts);
        } else {
            /* regular option */
            argv[argc++]=optStr;
        }
    }
    if (gDvmJni.useCheckJni) {                       //如果使用 CheckJni
        dvmUseCheckedJniVm(pVM);
    }

    if (gDvmJni.jniVm !=NULL) {
```

```
        dvmFprintf(stderr,"ERROR: Dalvik only supports one VM per process\n");
        return JNI_ERR;
    }
    gDvmJni.jniVm= (JavaVM* ) pVM;
/**创建 JNIEnv* /
    JNIEnvExt* pEnv= (JNIEnvExt* ) dvmCreateJNIEnv(NULL);

    gDvm.initializing= true;
/**启动虚拟机的各项服务,是虚拟机建立的关键* /
    std::string status=
            dvmStartup(argc,argv.get(),args->ignoreUnrecognized,(JNIEnv* )pEnv);
    gDvm.initializing= false;

    if (!status.empty()) {
        free(pEnv);
        free(pVM);
        LOGW("CreateJavaVM failed: %s",status.c_str());
        return JNI_ERR;
    }
    dvmChangeStatus(NULL,THREAD_NATIVE);              //改变状态
    * p_env= (JNIEnv* ) pEnv;
    * p_vm= (JavaVM* ) pVM;
    LOGV("CreateJavaVM succeeded");
    return JNI_OK;                                    //返回成功
}
```

Zygote 进程启动后会调用 JNI_CreateJavaVM()函数启动虚拟机来运行 Android 程序。现在就来分析一下虚拟机启动调用的 JNI_CreateJavaVM 的大致流程。一个用户进程只能创建一个虚拟机。首先函数创建虚拟机需要的 JavaVMExt 结构体,创建之后调用 memset 函数进行初始化,接着调用 dvmInitMutex()函数初始化信号量。然后函数把 JNI 的 args 转化为 argv,分析创建虚拟机时传入的函数参数,根据选项设定相应的值。然后为主线程创建一个 JNIEnv,做一些初始化的动作,需要调用本地代码。然后调用 dvmStartup 来初始化各个功能,其中包括启动垃圾回收、启动线程、启动内联本地方法、启动类、启动 JNI、初始化类等。在各种虚拟机的功能模块启动、初始化之后,虚拟机就算是启动完成,可以运行 Android 程序了。然后调用 dvmChangeStatus 函数改变线程当前状态为 THREAD_NATIVE,打印出"CreateJavaVM succeeded"信息,并返回 JNI_OK。

2. dvmCallJNIMethod 函数

dvmCallJNIMethod 在 dalvik/vm/Jni. cpp 中定义,源代码如下所示,函数流程图如图 3.5 所示。

代码清单 3.11　dalvik/vm/Jni. cpp:dvmCallJNI-Method()源代码

```
void dvmCallJNIMethod(const u4* args,JValue* pResult,const Method* method,Thread* self) {
    u4* modArgs= (u4* ) args;
```

```
    jclass staticMethodClass=NULL;

    u4 accessFlags=method->accessFlags;
                                        //赋值 accessFlags
    bool isSynchronized= (accessFlags & ACC_SYNCHRONIZED) !
=0;
    int idx=0;
    Object* lockObj;
/**检查被调用的函数是否为静态的*/
    if ((accessFlags & ACC_STATIC) != 0) {
      lockObj= (Object*) method->clazz;
      staticMethodClass= (jclass) addLocalReference(self,
(Object*) method->clazz);
    } else {
      lockObj= (Object*) args[0];
      /* add "this" */
      modArgs[idx++]= (u4) addLocalReference(self, (Object
*) modArgs[0]);
    }

    if (!method->noRef) {
        const char* shorty= &method->shorty[1];
/**处理被调用函数的各个属性*/
        while (*shorty != '\0') {          //循环检查方法的 shorty 字段
            switch (*shorty++) {
            case 'L':
                //LOGI(" local %d: 0x%08x",idx,modArgs[idx]);
                if (modArgs[idx] != 0) {
                    modArgs[idx]= (u4) addLocalReference(self, (Object*) modArgs[idx]);
                }
                break;
            case 'D':
            case 'J':
                idx++;
                break;
            default:
                /* Z B C S I--do nothing */
                break;
            }
            idx++;
        }
    }

    if (UNLIKELY(method->shouldTrace)) {
      logNativeMethodEntry(method,args);
    }
    if (UNLIKELY(isSynchronized)) {
```

开始 → 根据参数创建局部引用 → 改变线程状态为THREAD_NATIVE dvmChangeStatus → 调用相应方法 dvmPlatformInvoke → 改变线程状态为上一个状态 dvmChangeStatus → 转化引用结果 convertReferenceResult → 结束

图 3.5　dvmCallJNIMethod()流程图

```
        dvmLockObject(self,lockObj);
    }
    ThreadStatus oldStatus=dvmChangeStatus(self,THREAD_NATIVE);                    //改变状态
    ANDROID_MEMBAR_FULL();
    assert(method->insns !=NULL);
    JNIEnv* env=self->jniEnv;
    COMPUTE_STACK_SUM(self);                  //计算栈总数
/**调用函数*/
    dvmPlatformInvoke(env,
            (ClassObject*) staticMethodClass,
            method->jniArgInfo,method->insSize,modArgs,method->shorty,
            (void*) method->insns,pResult);                 //调用函数
    CHECK_STACK_SUM(self);
    dvmChangeStatus(self,oldStatus);                       //改变线程状态
    convertReferenceResult(env,pResult,method,self);
    if (UNLIKELY(isSynchronized)) {
        dvmUnlockObject(self,lockObj);
    }
    if (UNLIKELY(method->shouldTrace)) {
        logNativeMethodExit(method,self,*pResult);
    }
}
```

本函数首先遍历参数列表，创建合适的局部参数引用，并添加到引用列表中。调用dvmChangeStatus()函数改变线程状态为 THREAD_NATIVE。然后调用 dvmPlatformInvoke()函数根据参数调用本地函数，与底层进行交互。然后调用 dvmChangeStatus()函数将线程状态改为 oldStatus，也就是说改为上一个状态。接着调用 convertReferenceResult()函数，将引用结果进行转化，从局部引用或者全局引用转化为对象指针。到这里，调用 JNI 函数就完成了。

3. 函数 dvmResolveNativeMethod

dvmResolveNativeMethod 在 dalvik/vm/Native.cpp 中定义，源代码如代码清单 3.12 所示，函数流程图如图 3.6 所示。

代码清单 3.12　dalvik/vm/Native.cpp：dvmResolveNativeMethod()源代码

```
void dvmResolveNativeMethod(const u4* args,JValue* pResult,
    const Method* method,Thread* self)
{
    ClassObject* clazz=method->clazz;
/**是否为静态方法*/
    if (dvmIsStaticMethod(method)) {
        if (!dvmIsClassInitialized(clazz) && !dvmInitClass(clazz)) {
            assert(dvmCheckException(dvmThreadSelf()));
            return;
        }
    } else {
        assert(dvmIsClassInitialized(clazz) ||
```

```
                dvmIsClassInitializing(clazz));
        }
    /**在内部本地方法中查找要被调用的函数 * /
        DalvikNativeFunc infunc= dvmLookupInternalNativeMethod(method);
    /**如果找到将被调用的函数 * /
        if (infunc != NULL) {
            IF_LOGVV() {
                char * desc= dexProtoCopyMethodDescriptor(&method- > prototype);
                LOGVV("+ + + resolved native %s.%s %s,invoking",
                    clazz- > descriptor,method- > name,desc);
                free(desc);
            }
            if (dvmIsSynchronizedMethod(method)) {
                LOGE("ERROR: internal- native can't be declared 'synchronized'");
                LOGE("Failing on %s.%s",method- > clazz- > descriptor,method- > name);
                dvmAbort();                              // harsh,but this is VM- internal problem
            }
            DalvikBridgeFunc dfunc= (DalvikBridgeFunc) infunc;
    /**设置 Method 结构体的相应参数 * /
            dvmSetNativeFunc((Method * ) method,dfunc,NULL);
    /**调用函数 * /
            dfunc(args,pResult,method,self);
            return;
        }
    /**如果没有在内部本地函数中找到,那么在共享库中查找 * /
        void * func= lookupSharedLibMethod(method);
    /**如果查找到 * /
        if (func != NULL) {
    /**使用 JNI 调用桥 * /
            dvmUseJNIBridge((Method * ) method,func);
    /**跳转到被调用的函数 * /
            (* method- > nativeFunc) (args,pResult,method,self);
            return;
        }

        IF_LOGW() {
            char * desc= dexProtoCopyMethodDescriptor(&method- > prototype);
            LOGW("No implementation found for native %s.%s %s",
                clazz- > descriptor,method- > name,desc);
            free(desc);
        }
        dvmThrowUnsatisfiedLinkError(method- > name);
    }
```

本函数的作用是找到一个本地函数然后调用它。函数最开始获取方法的类 method→clazz,然后调用 dvmIsStaticMehod()函数判断当前的方法是否为静态方法,静态方法应该在类初始化之前被调用,如果是静态方法就检查类是否被初始化,如果没有被初始化,就初始化这

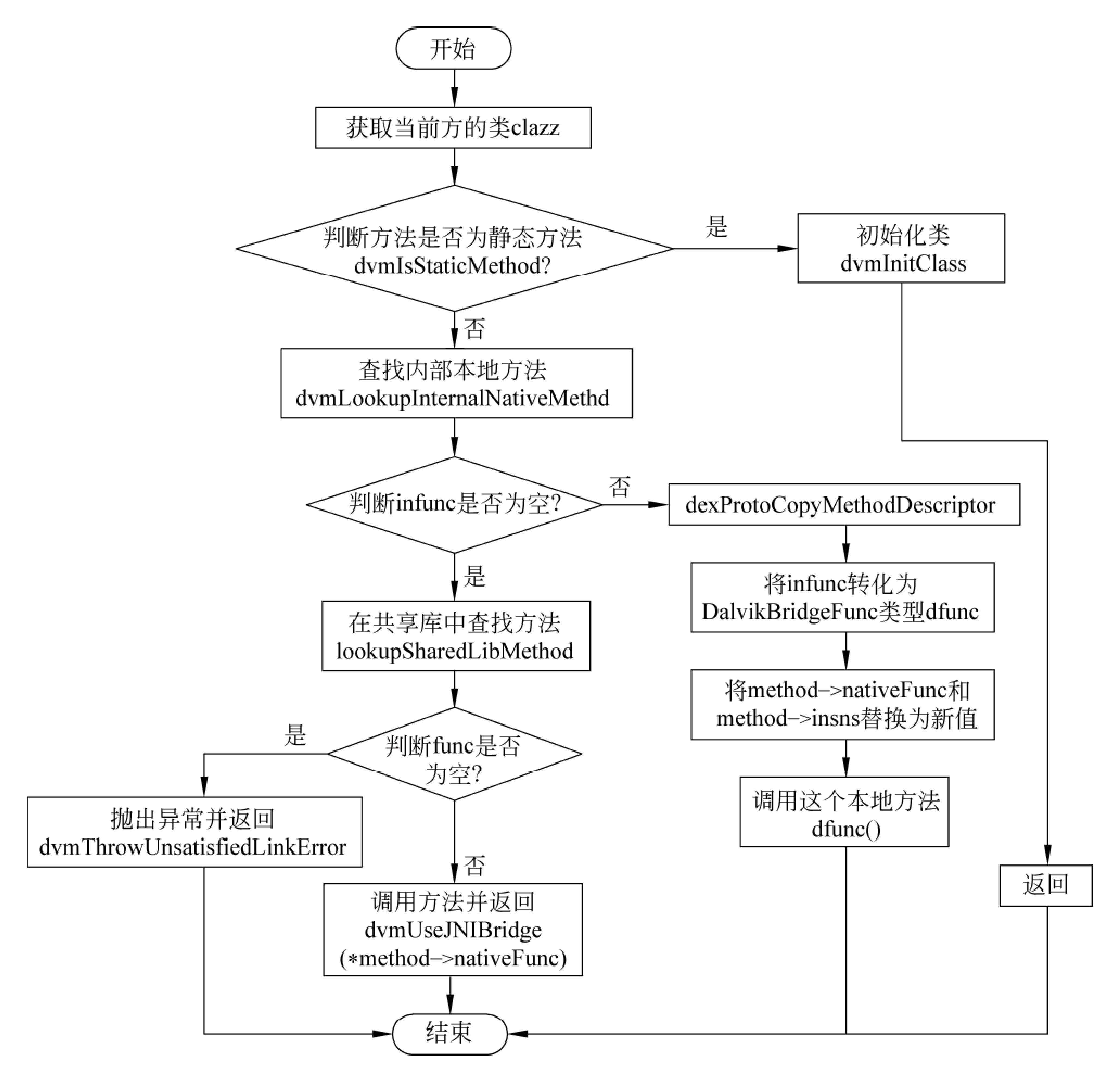


图 3.6 dvmResolveNativeMethod()函数流程图

个类,然后返回。如果不是静态方法,程序继续执行,调用 dvmLookupInternalNativeMethod 函数在内部本地方法列表中寻找这个方法,如果结果不为空,说明这个方法是虚拟机内部的本地方法,并且如果这个方法不是同步的方法,那么就调用 dvmSetNativeFunc()函数将 method→nativeFunc 和 method→insns 替换成新的值。然后调用这个本地方法,之后返回,完成对这个方法的处理。如果在虚拟机内部本地方法中没有找到这个方法,那么就在共享库中寻找这个本地方法,调用 lookupSharedLibMethod() 函数。如果找到本地方法,那么调用 dvmUseJNIBridge()函数使用 JNI Bridge,然后调用这个方法,完成对方法的处理。但是如果没有找到这个方法,就调用 dvmThrowUnsatisfiedLinkError()函数抛出异常,程序到此结束。

3.4.3 Java 调用 C 执行流程

分析 Dalvik 虚拟机执行本地函数(以 C 函数为例)的流程时,采用静态分析源代码函数的内部实现的同时还要配合 Linux 系统下的 GDB 动态调试跟踪工具来获得函数的调用情况和堆栈信息。

Dalvik 虚拟机在加载完类之后,执行类的 main 函数,首先会调用 JNI 本地函数接口 GetStaticMethodID(JNIEnv * env, jclass jclazz, const char * name, const char * sig)来获得主函数 main 的 MethodID,然后调用 JNI 接口函数 CallStaticVoidMethod(jclass clazz, jmethodID

methodID,…)来调用主函数。最后调用 Stack.cpp 文件中的 dvmCallMethodV(Thread * self, const Method * method, Object * obj, bool fromJni, JValue * pResult, va_list args)函数完成具体的函数调用功能。dvmCallMethodV 函数调用 callPrep 函数,callPrep 函数的主要功能是向解释器栈中压入一个栈帧,这里是主函数 main 的栈帧,并将 self→interpSave.curFrame 指针向前移动,函数返回 main 方法的类对象。向解释器栈中压入栈帧是由 dvmPushInterpFrame(Thread * self, const Method * method)函数完成的。由前述分析可知,dvmPushInterpFrame 函数需要压入两个栈帧,break frame 和方法本身的栈帧 regular frame。在 dvmPushInterpFrame 函数中维护了一个指针 stackPtr,在压栈过程中 stackPtr 指针指向栈的栈顶。开始时,判断 self→interpSave.curFrame 是否为空,如果为空表示当前方法栈为空,stackPtr 指针指向 self→interpStackStart,即解释栈的起始位置。如果 self→interpSave.curFrame 不为空,则 stackPtr 指向 self→interpSave.curFrame −1,即下一个栈帧起始的位置。压栈过程中需要的栈空间总数如下式所示。

```
stackReq=method->registersSize * 4              //方法参数和局部变量
        + sizeof(StackSaveArea) * 2             //break frame + regular frame
```

```
(gdb) p breakSaveBlock
$2 = (StackSaveArea *) 0x41da9fc0
(gdb) p *breakSaveBlock
$3 = {
  prevFrame = 0x41da9fe8,
  savedPc = 0x0,
  method = 0x0,
  xtra = {
    localRefCookie = 0,
    currentPc = 0x0
  },
  returnAddr = 0x4058e770
}
(gdb) p *saveBlock
$4 = {
  prevFrame = 0x41da9fd4,
  savedPc = 0x0,
  method = 0x41e5d250,
  xtra = {
    localRefCookie = 0,
    currentPc = 0x0
  },
  returnAddr = 0xfffffffd
}
```

图 3.7 栈帧结构体成员变量

由于寄存器为 32 位,所以保存一个寄存器需要 4B 的内存空间,一个 StackSaveArea 结构体大小为 20B,总共为栈帧分配 40B。接下来将栈指针 stackPtr 向前移动,为寄存器和栈帧分配内存空间。然后赋值栈帧的各个成员变量,初始化栈帧,self→interpSave.curFrame 指向方法栈帧的起始处。至此,类的主函数 main 已经被压入解释器的方法栈中。如图 3.7 所示为压入解释器栈的 break frame 和 regular frame 结构体的各个成员变量的内容。

如图 3.7 所示为解释器栈内存存储情况,可以看出解释器栈由高地址向低地址依次存放着 break frame 和 regular frame,栈帧的每个成员变量占有 4B。

然后会调用解释器到入口函数 dvmInterpret(Thread * self, const Method * method, JValue * pResult)执行 main 方法,解释器开始执行 main 方法的字节码。下面来详细分析一下虚拟机执行本地函数的过程,如图 3.8 所示,为 Java 代码调用本地代码的流程。

无论是本地函数还是 Java 函数,方法的属性都存储在 Method 结构体中。Method 结构体使用一个 4B 整型变量 accessFlags 来表示方法的访问属性。在文件 androidsource/dalvik/libdex/DexFile.h 中定义了各种方法属性和对应的十六进制数值,比如,ACC_NATIVE=0x00000100 表示本地方法的 accessFlags 值为 0x00000100,也就是十进制的 256。虚拟机可以通过判断 accessFlags 字段的值来判断方法的访问属性,包括 public、private、protected、static、final 等。

当虚拟机要执行一个本地函数时,会首先执行 Method→nativeFunc。nativeFunc 是 Method 结构体的一个成员字段,是一个 DalvikBridgeFunc 类型的函数指针,初始时为 void dvmResolveNativeMethod(const u4 * args, JValue * pResult, const Method * method, Thread * self),Method 结构体的另一个字段 insns 是一个 2B 的指针,指向这个函数的实际

```
(gdb) x/18xw 0x41da9f8c
0x41da9f8c:   0x41da9fd4   0x00000000   0x41e5d250   0x00000000
0x41da9f9c:   0xfffffffd   0x41da9fd4   0x00000000   0x41e280e8
0x41da9fac:   0x426c8a5c   0x41e1efa8   0x00000009   0xfffffffd
0x41da9fbc:   0x405a18c0   0x41da9fe8   0x00000000   0x00000000
0x41da9fcc:   0x00000000   0x4058e770
```

图 3.8　解释器栈内存情况

代码位置，也就是调用加载的动态链接库中函数的机器码在内存中的地址。刚开始时，虚拟机需要执行 Method→nativeFunc，也就是 androidsource/dalvik/vm/Native.cpp 文件中的 dvmResolveNativeMethod 函数来处理本地方法，处理的主要工作就是查找本地函数，然后调用执行它。dvmResolveNativeMethod 函数首先调用 dvmLookupInternalNativeMethod 函数在 Dalvik 虚拟机的内部本地方法中查找这个本地函数，对于虚拟机的内部的类的本地方法，可以通过这个方法找到。而对于用户编写的应用程序中的本函数来说，使用这个函数是找不到的，这就需要继续执行，调用 lookupSharedLibMethod 函数在虚拟机的本地动态链接库中查找函数。虚拟机全局结构体 gDvm 中的 nativeLibs 字段是一个指向哈希表类型的指针，用来表示虚拟机本地共享库表。对于 nativeLibs 中的每一个共享库入口调用 findMethodInlib 函数在这个本地动态链接库中查找本地函数，直到找到这个本地函数，返回指向它的指针，如图 3.9 所示为 Java 代码调用本地代码流程。

Method 结构体的两个成员变量 nativeFunc 和 insns 有三个基本状态：

(1)（初始状态）nativeFunc＝dvmResolveNativeMethod，insns＝NULL；

(2)（内部本地方法）nativeFunc＝<impl>，insns＝NULL；

(3)（JNI）nativeFunc＝JNI call bridge，insns＝<impl>。

最普遍的状态转换是(1)到(2)和(1)到(3)。如果调用 dvmLookupInternalNativeMethod 函数查找到一个函数是虚拟机内部本地函数，就调用 dvmSetNativeFunc 函数将这个本地函数的 nativeFunc 字段设置为指向这个本地函数的实现代码的指针，insns 成员为 NULL，也就是(1)到(2)的状态转换，然后调用这个本地函数。如果这个函数不是虚拟机内部本地函数，那么调用 lookupSharedLibMethod 函数找到本地函数后，调用 dvmUseJNIBridge 函数使 nativeFunc 指向 JNI Call Bridge，insns 指向函数的代码实现位置，这就是(1)到(3)的状态转换。如果 gDvmJni.useCheckJni 为真，那么 JNI call bridge 为 dvmCheckCallJNIMethod，否则为 dvmCallJNIMethod。

对于用户应用程序中的本地函数，虚拟机调用找到本地函数后，调用 nativeFunc 指向的 JNI Call bridge，关键函数是 dvmCallJNIMethod。dvmCallJNIMethod 函数调用了 androidsource/dalvik/vm/arch/arm/CallEABI.S 文件的 dvmPlatformInvoke 函数来执行本地方法，dvmPlatformInvoke 是用汇编编写实现的函数。JNI Call bridge 可以说是解释的 Java 代码和本地函数代码之间的一个桥梁。如下所示为 dvmPlatformInvoke 函数中从 Java 代码跳转至本地代码的关键语句。

```
ldmia r9,{r2-r3}                @ r2/r3<-argv[0]/argv[1]
ldr    ip,[fp,#8+FP_ADJ]        @ ip<-func
blx    ip                       @ call func
```

将函数的参数传递给 r2 和 r3 寄存器，然后本地函数的指针传递给 ip 寄存器，执行 blx 跳转语句，程序跳转至 ip 寄存器[10]指向的地址处执行，即跳转至本地动态链接库的本地代码中执行。

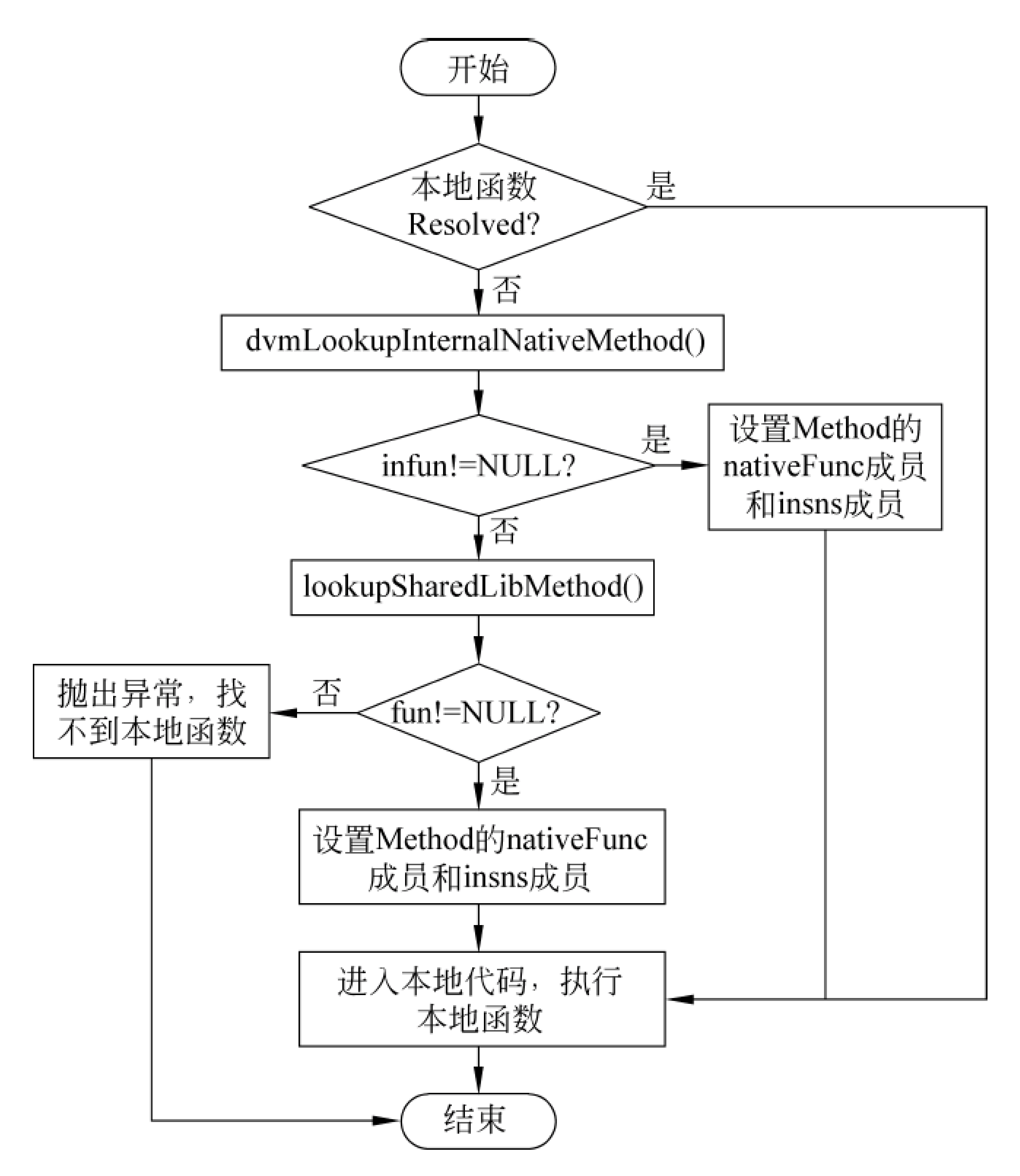


图 3.9 Java 代码调用本地代码流程

3.5 C 调用 Java 执行流程分析

3.5.1 本地调用接口函数结构体

本地代码通过调用 JNI 接口函数来访问 Dalvik 虚拟机的特性。可以通过使用一个接口指针来获得 JNI 接口函数，一个接口指针是一个指向指针的指针。typedef const struct JNINativeInterface * JNIEnv 定义了一个指向 JNINativeInterface 结构体的指针 JNIEnv。JNINativeInterface 这个结构体中的成员全部都是函数指针，这些函数就是所谓的本地调用接口函数，即 JNI 函数。每一个成员函数指针都指向一个接口函数。如图 3.10 所示为接口指针的组织关系。

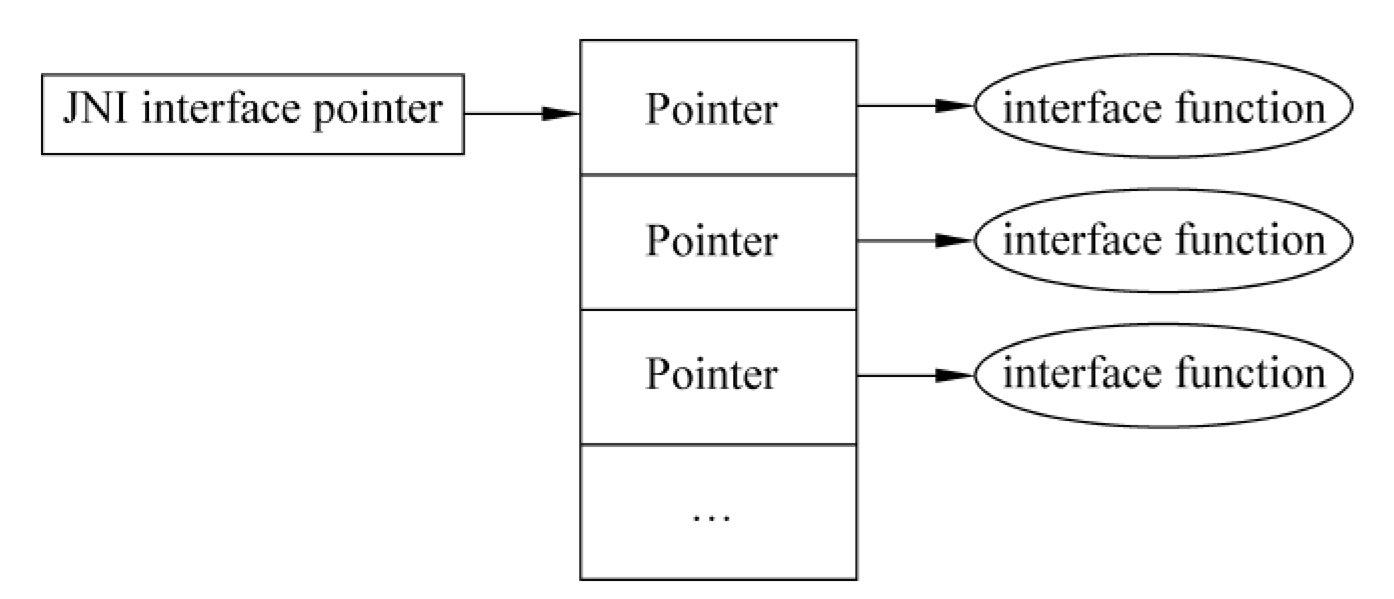


图 3.10 接口指针函数表结构

一个虚拟机可以提供多种版本的 JNI 函数接口表，也就是说可以定义多个指向 JNINativeInterface 类型的指针变量，每一个变量都是一个函数接口表。例如，在 Android

系统的 Dalvik 虚拟机中，就存在以下两个版本的 JNI 函数接口表。

(1) 一个接口函数表进行十分周密的合法参数检查，确保函数的返回值和参数合法，适合于进行调试，函数表的定义在文件 android/dalvik/vm/CheckJni.cpp 中；

(2) 另一个接口函数表只进行最少数量的 JNI 检查工作，所以使用起来较为高效，这个函数表定义在文件 android/dalvik/vm/Jni.cpp 中。

另外，JNI 接口函数指针只在当前的线程中有效。所以，一个本地方法不能够将一个接口指针从一个线程传递到另一个线程。本地函数执行时，JNI 接口指针作为函数参数传递给本地函数，一个本地函数可以被来自于不同 Java 线程的函数调用，所以它可以接收不同的 JNI 接口函数指针作为函数参数。

3.5.2 关键函数

1. 函数 RegisterNatives

RegisterNatives 在 dalvik/vm/Jni.cpp 中定义，源代码如下所示，函数流程图如图 3.11 所示。

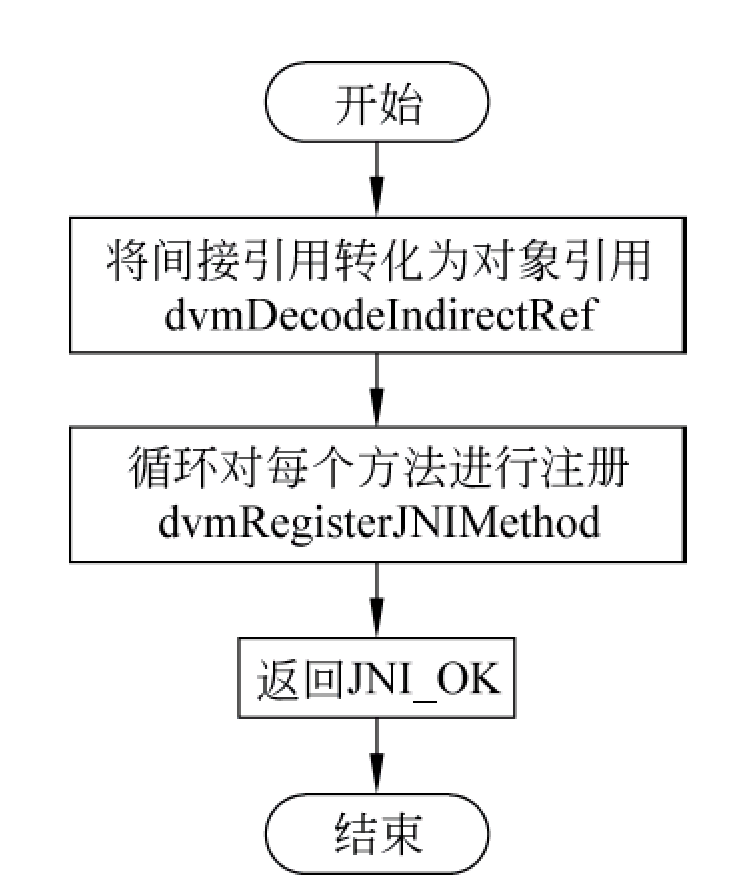


图 3.11 RegisterNatives()函数流程图

代码清单 3.13 dalvik/vm/Jni.cpp：RegisterNatives()源代码

```
static jint RegisterNatives(JNIEnv* env,jclass jclazz,const JNINativeMethod* methods,jint nMethods)
{
    ScopedJniThreadState ts(env);
    ClassObject* clazz= (ClassObject*) dvmDecodeIndirectRef(ts.self(),jclazz);
    //找到类对象
    if (gDvm.verboseJni) {
        LOGI("[Registering JNI native methods for class %s]",
            clazz->descriptor);
    }
    for (int i=0; i<nMethods; i++) {                              //循环注册每一个本地方法
        if (!dvmRegisterJNIMethod(clazz,methods[i].name,
                methods[i].signature,methods[i].fnPtr))           //注册本地方法
        {
            return JNI_ERR;
        }
    }
    return JNI_OK;                                                //返回成功
}
```

RegisterNatives()函数是 JNI 机制中实现的重要函数，负责实现注册本地函数的功能。函数体首先调用 dvmDecodeIndirectRef()函数对间接引用进行转化，转化为类对象 clazz，然后在 for 循环中对 methods 本地方法数组中的每一个本地方法进行注册，调用函数 dvmRegisterJNIMethod()，函数参数为方法的名字、签名和函数指针，如果注册失败，函数返回 JNI_ERR，否则注册成功，返回 JNI_OK。

2. 函数 dvmRegisterJNIMethod

dvmRegisterJNIMethod 在 dalvik/vm/Jni.cpp 中定义，源代码如下所示，函数流程图如图 3.12 所示。

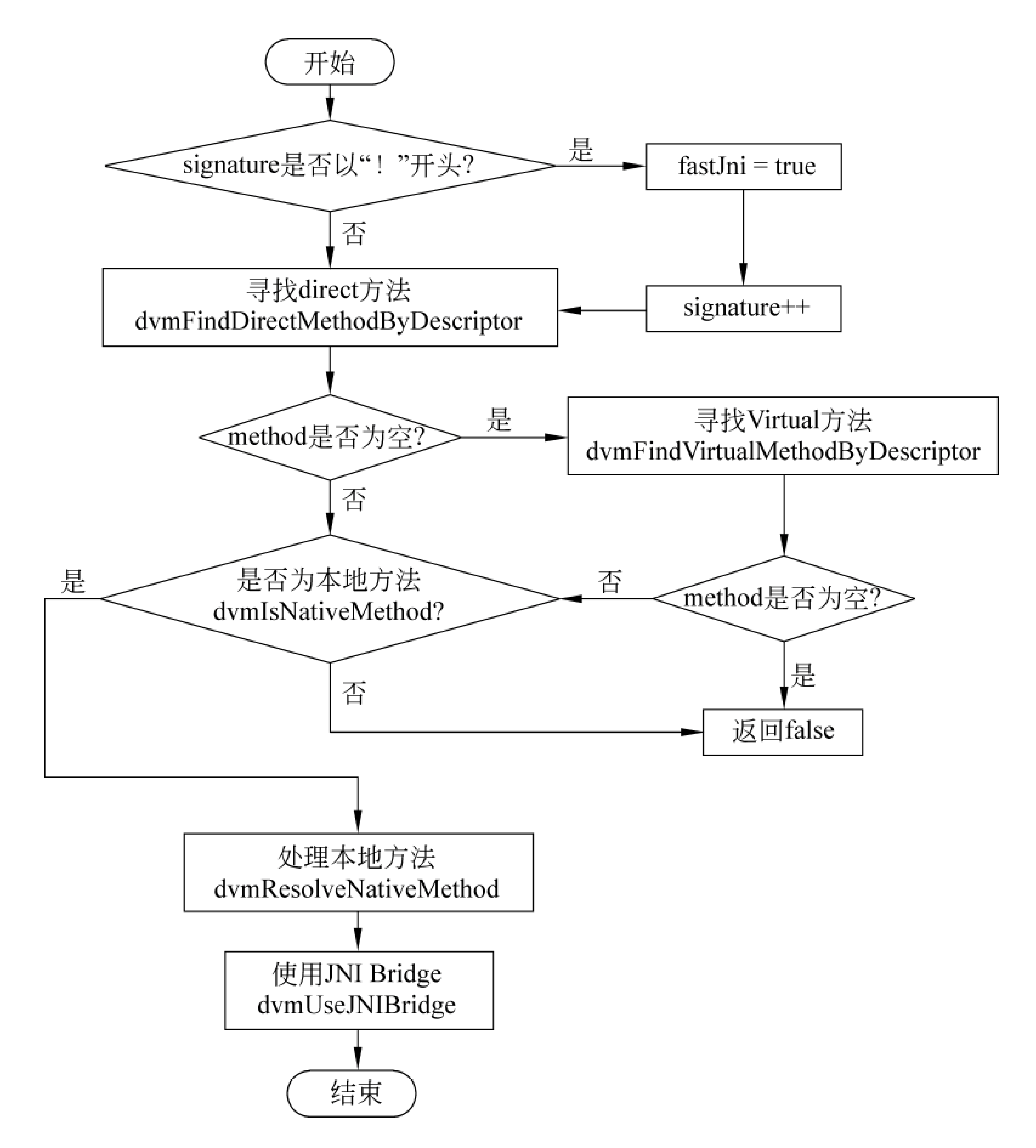


图 3.12 dvmRegisterJNIMethod()函数流程图

代码清单 3.14 dalvik/vm/Jni.cpp：dvmRegisterJNIMethod()源代码

```
static bool dvmRegisterJNIMethod(ClassObject* clazz,const char* methodName,const char* signature,
void* fnPtr)
{
    if (fnPtr==NULL) {
        return false;
    }

      bool fastJni=false;
/**处理函数签名*/
    if (*signature=='!') {
        fastJni=true;
        ++signature;
        LOGV("fast JNI method %s.%s:%s detected",clazz->descriptor,methodName,signature);
    }
```

```
/**根据签名查找 direct 方法 */
    Method* method=dvmFindDirectMethodByDescriptor(clazz,methodName,signature);
/**如果没有找到,那么根据签名查找 virtual */
    if (method==NULL) {
        method=dvmFindVirtualMethodByDescriptor(clazz,methodName,signature);
    }
/**如果没有找到,返回错误信息 */
    if (method==NULL) {
        dumpCandidateMethods(clazz,methodName,signature);
        return false;
    }
/**如果不是本地方法,不能注册,打印错误信息 */
    if (!dvmIsNativeMethod(method)) {
        LOGW("Unable to register: not native: %s.%s:%s",clazz->descriptor,methodName,signature);
        return false;
    }

    if (fastJni) {
            if (dvmIsSynchronizedMethod(method)) {
                    LOGE("fast JNI method %s.%s:%s cannot be synchronized",
                   clazz->descriptor,methodName,signature);
            return false;
        }
/**如果不是静态方法,打印错误信息 */
        if (!dvmIsStaticMethod(method)) {
            LOGE("fast JNI method %s.%s:%s cannot be non-static",
                   clazz->descriptor,methodName,signature);
            return false;
        }
    }
/**处理本地方法 */
    if (method->nativeFunc !=dvmResolveNativeMethod) {
        LOGV("Note: %s.%s:%s was already registered",clazz->descriptor,methodName,signature);
    }

    method->fastJni=fastJni;
/**使用 JNI 桥 */
    dvmUseJNIBridge(method,fnPtr);

    LOGV("JNI-registered %s.%s:%s",clazz->descriptor,methodName,signature);
    return true;
}
```

函数的参数为 ClassObject * clazz，const char * methodName，const char * signature，void * fnPtr。首先判断 fnPtr 是否为空，如果为空，则返回 false。然后判断函数的签名的第一位是否为“!”，若为“!”，表示本地代码不需要额外的 JNI 参数。然后方法列表中按照方法描述符寻找 direct 方法，若找到则在方法列表中查找 virtual 方法，若没有找到则返回

false。查找到方法之后，调用 dvmIsNativeMethod 检查方法是否为本地方法，若不是本地方法则打印信息并返回 false。若为本地方法则对方法进行处理，注册。接着调用 dvmUseJNIBridge()函数，注册成功后返回 true。

3. 函数 dvmLoadNativeCode

dvmLoadNativeCode 在 dalvik/vm/Native.cpp 中定义，源代码如下所示，函数流程图如图 3.13 所示。

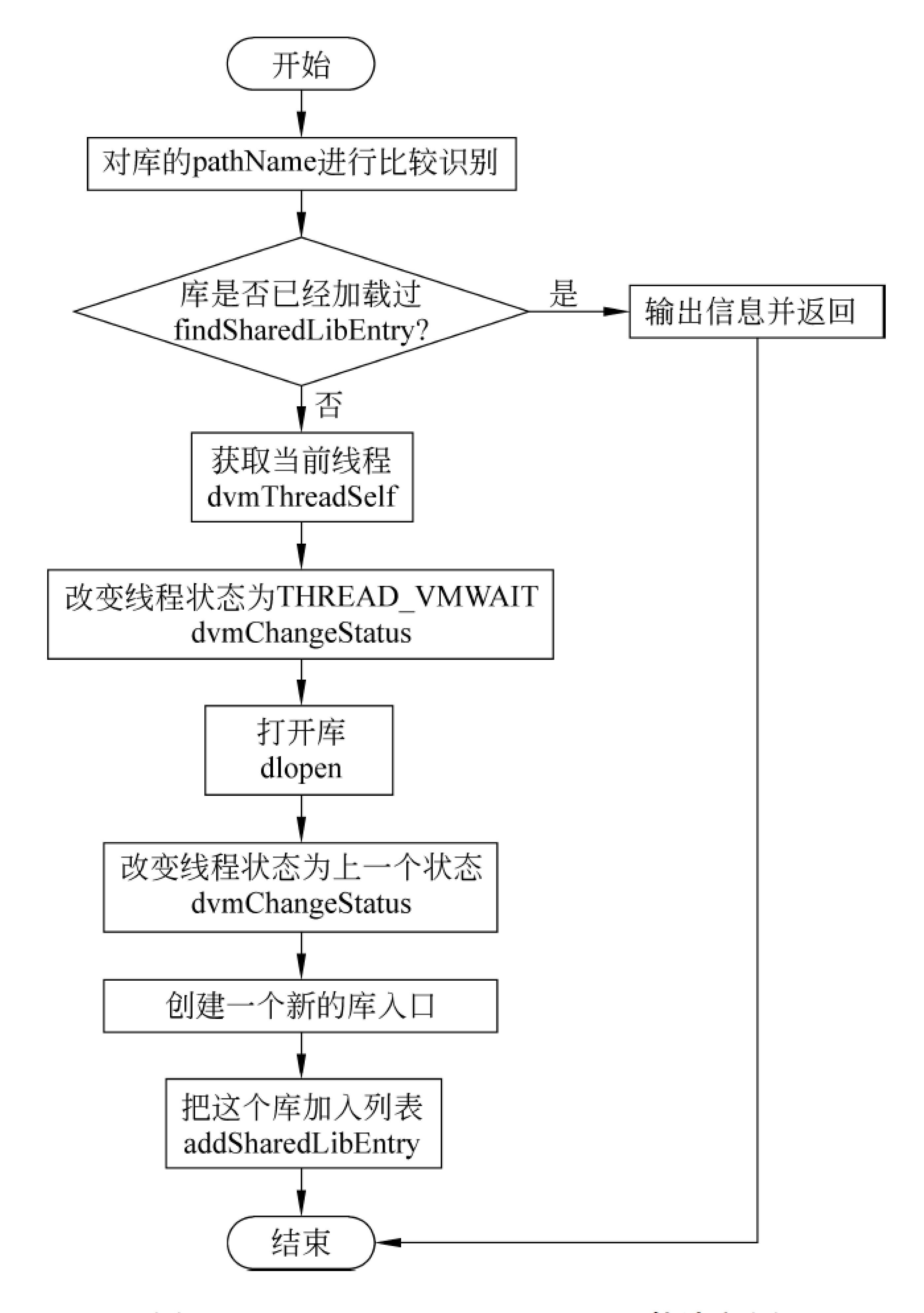


图 3.13 dvmLoadNativeCode()函数流程图

代码清单 3.15 dalvik/vm/Native.cpp：dvmLoadNativeCode()源代码

```
bool dvmLoadNativeCode(const char* pathName,Object* classLoader,char** detail)
{
    SharedLib* pEntry;
    void* handle;
    bool verbose;

    verbose=!!strncmp(pathName,"/system",sizeof("/system")-1);            //检查路径
    verbose=verbose && !!strncmp(pathName,"/vendor",sizeof("/vendor")-1);
```

```
    if (verbose)
        LOGD("Trying to load lib %s %p",pathName,classLoader);

    *detail=NULL;
/**查找共享库的入口 */
        pEntry=findSharedLibEntry(pathName);
    if (pEntry !=NULL) {
        if (pEntry->classLoader !=classLoader) {
            LOGW("Shared lib '%s' already opened by CL %p; can't open in %p",
                pathName,pEntry->classLoader,classLoader);
            return false;
        }
        if (verbose) {
            LOGD("Shared lib '%s' already loaded in same CL %p",
                pathName,classLoader);
        }
        if (!checkOnLoadResult(pEntry))
            return false;
        return true;
    }
/**获取当前的线程 */
    Thread* self=dvmThreadSelf();
    ThreadStatus oldStatus=dvmChangeStatus(self,THREAD_VMWAIT);
    handle=dlopen(pathName,RTLD_LAZY);
/**改变线程状态 */
    dvmChangeStatus(self,oldStatus);

    if (handle==NULL) {
        *detail=strdup(dlerror());
        return false;
    }

        SharedLib* pNewEntry;
/**分配共享库的内存空间,并初始化结构体的成员变量 */
    pNewEntry=(SharedLib*) calloc(1,sizeof(SharedLib));
    pNewEntry->pathName=strdup(pathName);
    pNewEntry->handle=handle;
    pNewEntry->classLoader=classLoader;
    dvmInitMutex(&pNewEntry->onLoadLock);
    pthread_cond_init(&pNewEntry->onLoadCond,NULL);
    pNewEntry->onLoadThreadId=self->threadId;
/**添加共享库的入口 */
    SharedLib* pActualEntry=addSharedLibEntry(pNewEntry);

    if (pNewEntry !=pActualEntry) {
        LOGI("WOW: we lost a race to add a shared lib (%s CL=%p)",
            pathName,classLoader);
```

```
/**释放共享库内存资源 * /
        freeSharedLibEntry(pNewEntry);
        return checkOnLoadResult(pActualEntry);
    } else {
        if (verbose)
            LOGD("Added shared lib %s %p",pathName,classLoader);

        bool result=true;                                           //结果为真
        void* vonLoad;
        int version;

        vonLoad=dlsym(handle,"JNI_OnLoad");
        if (vonLoad==NULL) {
            LOGD("No JNI_OnLoad found in %s %p,skipping init",
                pathName,classLoader);
        } else {
             OnLoadFunc func=(OnLoadFunc)vonLoad;
            Object* prevOverride=self->classLoaderOverride;

            self->classLoaderOverride=classLoader;
            oldStatus=dvmChangeStatus(self,THREAD_NATIVE);          //改变线程状态
            if (gDvm.verboseJni) {
                LOGI("[Calling JNI_OnLoad for \"%s\"]",pathName);
            }
            version=(*func)(gDvmJni.jniVm,NULL);
            dvmChangeStatus(self,oldStatus);                        //改变线程状态
            self->classLoaderOverride=prevOverride;

            if (version != JNI_VERSION_1_2 && version != JNI_VERSION_1_4 &&
                version != JNI_VERSION_1_6)                         //检查 JNI 版本号
            {
                LOGW("JNI_OnLoad returned bad version (%d) in %s %p",
                    version,pathName,classLoader);
                result=false;
            } else {
                if (gDvm.verboseJni) {
                    LOGI("[Returned from JNI_OnLoad for \"%s\"]",pathName);
                }
            }
        }

        if (result)
            pNewEntry->onLoadResult=kOnLoadOkay;
        else
            pNewEntry->onLoadResult=kOnLoadFailed;

        pNewEntry->onLoadThreadId=0;
```

```
            dvmLockMutex(&pNewEntry->onLoadLock);                    //锁住信号量
            pthread_cond_broadcast(&pNewEntry->onLoadCond);
            dvmUnlockMutex(&pNewEntry->onLoadLock);                  //解锁信号量
            return result;
        }
    }
```

本函数功能是加载指定绝对路径的本地代码。如果已经加载过这个库，程序不执行任何操作直接返回。首先程序通过字符串比较识别路径 pathName，然后在已知哈希表中查找当前库，看是否已经加载过，如果加载过则直接返回，否则就会执行程序以加载共享库。首先调用 dvmThreadSelf()获得当前线程，然后改变线程状态为 THREAD_VMWAIT，然后调用 dlopen()函数打开这个动态链接库，再把线程状态改为上一个线程状态。然后声明一个 SharedLIb * 类型的共享库入口，然后将当前的库的信息赋值给它，创建一个信息共享库入口。然后调用 addSharedLibEntry()函数把这个入口添加到共享库列表中，到这里程序就将新的共享库加载到虚拟机中了，以供其他程序调用共享库中的本地函数。

3.5.3　C 调用 Java 执行流程

上述为 Java 代码调用 C 代码的流程，在 Java 函数调用执行 C 函数的过程中，本地调用机制保证 C 函数可以调用 Java 函数。此时，C 函数使用了 Java 类的资源。C 函数方法 Java 类中的资源需要借助 JNI 本地调用接口函数来完成，例如，若要访问 Java 类的变量，在 C 函数中需首先调用 JNI 接口函数 GetFieldID 或 GetStaticFieldID(静态变量)来获得变量的 FieldID，然后调用 JNI 接口函数 GetTypeField 并传入 FieldID 参数来获得具体的变量，其中“Type”为变量的类型，可以是 Int、Long、Double 等类型；若要访问 Java 类的函数，首先调用 JNI 接口函数 GetMethodID 或者 GetStaticMethodID(静态方法)来获取函数的 MethodID，然后调用 JNI 接口函数 CallTypeMethod 并传入 MethodID 参数来调用。

Java 函数由解释器执行。解释器执行完 Java 函数后返回至本地动态链接库继续执行本地函数，如图 3.14 所示为本地代码调用 Java 代码示意图。

在本地代码中，通过偏移量来定位本地调用接口函数表中的函数，如下代码片段为本地函数调用本地接口函数 GetMethodID 的语句。

```
ldr.w   r5,[r2,#136]      ;0x88
blx     r5
```

第 1 句取得 GetMethodID 函数的地址，一个地址占有 4B，136 除以 4 等于 34，而本地调用接口函数表中第 34 个接口函数就是 GetMethodID 函数。取得的地址存放在 r5 寄存器中，然后第 2 句跳转到 GetMethodID 函数的地址处开始执行，调用 GetMethodID 函数。

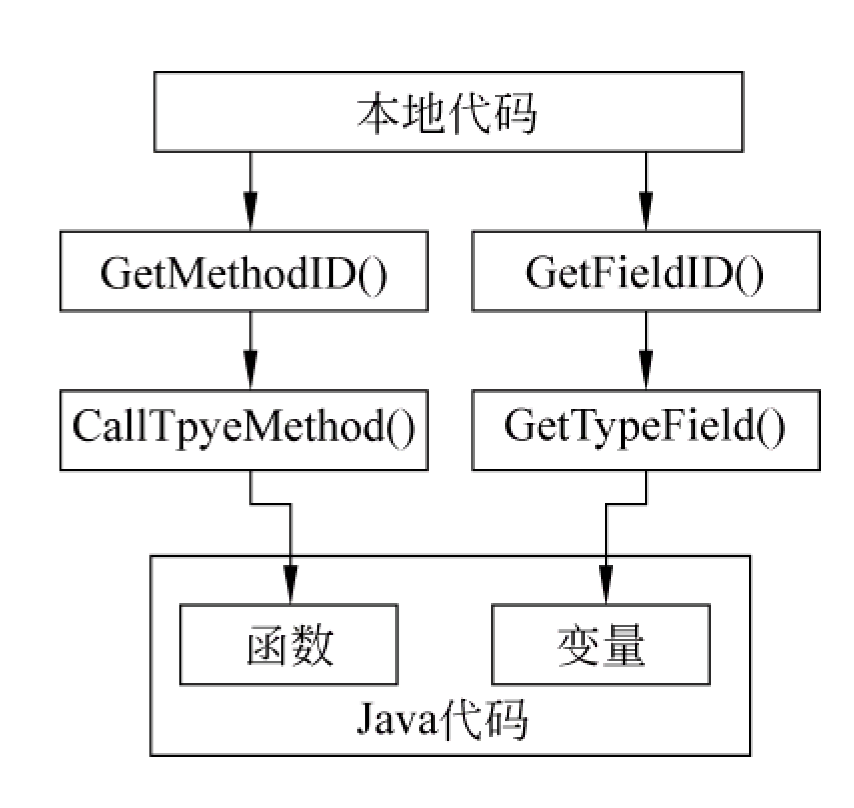


图 3.14　本地代码调用 Java 代码示意图

代码清单 3.16　手动编写待调用的 Java 源代码

```
package loda.Test8d;
```

```
import android.app.Activity;
import android.util.Log;
import android.os.Bundle;

public class Test8d extends Activity {

    private native String HelloWorld(int a,String b,double c);          //声明本地函数
    static {
        System.loadLibrary("test8d");                                    //加载本地库
    }
    private String CallFromJNI(int a,String b,double c)                 //Java 函数定义
    {
        Log.i("loda","int:"+ a+ "_String:"+ b+ "_double:"+ c);
        return "From CallFromJNI in Java";
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.i("loda","libtest8d Before Call JNI HelloWorld");
    String vStr= "It is a string from Java";
    String vRet= HelloWorld(5799232,vStr,123.456);                       //调用本地函数
    Log.i("loda",vRet);
    Log.i("loda","libtest8d After Call JNI HelloWorld");
    setContentView(R.layout.main);
    }
}
```

如代码清单 3.16 所示为使用 C 代码调用 Java 代码的例子中 Java 代码的内容，Java 代码中定义了 Java 函数 CallFromJNI，在 C 代码中要调用 Java 代码中的 CallFromJNI 函数，首先调用 JNI 接口函数（* env）→GetMethodID 来取得 Java 函数 CallFromJNI 的 MethodID，然后调用接口函数（* env）→CallObjectMethod 来调用指定 MethodID 的 Java 函数，即完成了 C 代码对 Java 代码的调用。C 代码清单如下所示。

代码清单 3.17 手动编写调用 Java 的 C 源代码

```
#include<jni.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define LOG_TAG "C LOG DEMO"
#undef LOG
#include<utils/Log.h>
char gTest8Buffer[]= "Unicode based Global Sring Reference from JNI- C";
JNIEXPORT jstring JNICALL Java_loda_Test8d_Test8d_HelloWorld(JNIEnv * env,jobject jobj,jint test_int,
jstring test_string,jdouble test_double)          //本地函数实现
{
```

```
    jclass vClass= (* env)->GetObjectClass(env,jobj);
    jmethodID vMethodID= (* env)->GetMethodID(env,vClass,"CallFromJNI","(ILjava/lang/String;D)Ljava/
lang/String;");                                          //查找 Java 函数的 MethodID
    if(vMethodID==NULL)
    {
        LOGI("Falied to load JavaFunc CallFromJNI");
        return NULL;
    }
    jstring vStr= (* env)->NewStringUTF(env,gTest8Buffer);
jstringvRetStr= (* env)->CallObjectMethod(env,jobj,vMethodID,123,vStr,4321.1234);
                                   //调用 Java 函数
    const jbyte * jni_string= (* env)->GetStringUTFChars(env,vRetStr,0);
    if(jni_string==NULL)
    {
        return NULL;
    }
     LOGI("Java Method Return String:%s",jni_string);
     (* env)->ReleaseStringUTFChars(env,vRetStr,jni_string);//释放字符串资源
return vStr;
```

小　　结

本章主要分析了 Android 系统 Dalvik 虚拟机的本地调用机制的实现原理和工作流程。通过实际编程实践，总结了使用本地调用机制编程的主要步骤和过程；通过分析源代码得到了 Dalvik 虚拟机建立本地调用机制环境的过程和调用的主要函数，以及 Java 代码和 C 代码互相调用的详细过程，更加微观地描述了本地调用机制的工作流程。

第 4 章 反射机制模块的原理及实现

本章内容提要

- ✍ 什么是反射机制？
- ✍ 反射机制有什么作用？
- ✍ 反射机制的内部函数结构是什么样的？
- ✍ 反射机制是如何实现的？
- ✍ 反射机制有什么优点和不足？

反射机制是 Dalvik 虚拟机中的核心机制，也是 Android 开发中不可缺少的重要工具。本章将围绕以上 5 点对 Dalvik 虚拟机中的反射机制进行细致的分析。首先概述什么是反射机制，以及反射机制在 Dalvik 虚拟机中和 Android 平台上的地位和作用。随后对反射机制的实现原理进行深入剖析，详细分析反射机制内部的核心类和函数，总结归纳反射机制的具体实现流程。由于反射机制与 Dalvik 虚拟机中的其他机制和模块不同，它更像是一个便于 Android 开发的工具，所以本章最后一节将介绍 Android 开发人员在开发具体应用时是如何使用反射机制的。

4.1 概述

在计算机领域中，反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通过采用某种机制来实现对自己行为的描述（self-representation）和监测（examination），并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

同时，反射机制是 Java 被当作动态（或准动态）语言的一个关键性质。反射机制允许程序在运行的过程中通过反射机制的 API 取得任何一个已知名称的类的内部信息，包括其中的描述符、超类，也包括属性和方法等所有信息，并且可以在程序运行时改变属性的相关内容或调用其内部的方法。合理熟练地运用反射机制可以让 Java 程序变得更加灵活，让程序得到更新而不用改动原有的方法代码的目标得以实现。另外，类反射机制在数据库方面的应用更是非常广泛。同时，反射机制具有很好的派生性，例如，Java 语言基于类反射技术还开发提供了动态代理和元数据两种机制，使程序逻辑更加简单安全性更高。对于这两种机制，会在后面的章节中具体介绍它们的实现原理以及功能架构。

4.2　反射机制实现代码示例

反射机制和 Dalvik 虚拟机中的其他模块和机制不同，它更像是一种方便程序员编写 Java 程序使程序更加灵活简洁的工具。所以不但要了解反射机制的实现原理和内部结构，还要学习如何使用反射机制来编写更加灵活的程序。首先通过具体的示例来形象地展示什么是反射机制，反射机制在具体程序中是如何应用的以及反射机制的作用是什么。

代码清单 4.1　手动编写：forName.java 源代码

```
import java.lang.reflect.Method;
import java.util.Random;
class simple_00{
public String get_0() {
    return "0";
}
public String get_1() {
    return "1";
}
public String get_2() {
    return "2";
}
public String get_3() {
    return "3";
}
public String get_4() {
    return "4";
}
};
class simple_01{
public String get_0() {
    return "0";
}
public String get_1() {
    return "1";
}
public String get_2() {
    return "2";
}
public String get_3() {
    return "3";
}
public String get_4() {
    return "4";
}
```

```
};
class simple_02{
public String get_0() {
     return "0";
}
public String get_1() {
     return "1";
}
public String get_2() {
     return "2";
}
public String get_3() {
     return "3";
}
public String get_4() {
     return "4";
}
};
public final class test_1 {
          public void run(int cnt) {
               Random randomGenerator= new Random();
               int cidx;
               for (int i= 0; i< cnt; i++) {
                    cidx= randomGenerator.nextInt(3);
                    try {
Class<?> clazz= ClassClass.forName("simple_0"+ Integer.toString(cidx));
                    Method method= clazz.getDeclaredMethod("get_0");
                    method.invoke(clazz.newInstance());
                  } catch (Exception ioe) {
                    System.out.println(ioe);
                  }
             }
        }
        public static void main (final String[] args) {
             long time_before= System.currentTimeMillis();
             test_1 test= new test_1();
             test.run(10000);
             long time_after= System.currentTimeMillis();
             long different= time_after - time_before;
             System.out.println(different);
        }
        }
```

首先这段代码定义了三个类 simple_00、simple_01 和 simple_02，然后在每个类中又定义了 5 个方法 String get_0()、String get_1()、String get_2()、String get_3()以及 String

get_4(),这些方法的作用是分别返回结果 0、1、2、3、4。随后程序运用 Random()函数产生了一个随机数,并且赋值给 cidx,并且运用反射机制中常见的方法 Class. forName 获得指定名称的类并输出结果。关于 Class. forName 的具体实现流程在后面的章节中将详细讲述。

点拨　在程序员编写一个 Android 程序时需要用到许多私有方法,这些私有方法不能被直接调用,比如 endCall 方法。这个方法的主要作用是自动挂断电话,在编写黑名单管理应用的时候程序员必须调用这个方法才能对指定的电话号进行拦截挂断。但是 endCall 方法在 Android 1.5 以后被设置为私有方法了。所以如果要实现对指定电话自动挂断的功能就一定要用到反射机制。下面将介绍如何在具体应用中运用反射实现自动挂断电话的功能。

首先利用反射机制中的 Class. forName 方法获得 ServiceManager,ServiceManager 是 Android 平台中的一个基本模块,它是用来管理 Android 系统中的 service,如 InputMethodService、ActivityManagerService 等服务。再通过反射机制中的 getMethod 方法调用服务管理模块中的 getService 方法得到 telephone 的 IBinder,再将 IBinder 转化为 ITelephony 接口。ITelephony 是 Android 系统中基本程序框架功能之一,主要提供和电话相关的 API 交互。至此,程序获得了 ITelephony 类中最为关键的 endCall()方法,在需要的部分即可调用 endCall()方法来实现自动挂断电话的功能。

代码清单 4.2　lanjie. java 源代码

```
if (act.equals(PHONE_STATE)) {
            String num=bundle.getString("incoming_number");
//判断来电,并显示文本
            Toast.makeText(context,"有来电",Toast.LENGTH_LONG)
                .show();
            if (num != null && num.length() > 0) {
                FilterDB fdb= new FilterDB(context);
                PhoneDB phoneItem= fdb.getPhoneItem(num);
                if (phoneItem != null && phoneItem.isPhoneDisabled()) {
//调用 stopCall()函数对指定来电进行拦截
                    Toast.makeText(context,"拦截来电",Toast.LENGTH_LONG)
                    .show();
                    stopCall();
                    Toast.makeText(context,"拦截完毕",Toast.LENGTH_LONG)
                    .show();
                    FilterIncoming in= new FilterIncoming();
                    in.setDate(new Date());
                    in.setItemId(phoneItem.getId());
                    in.setNum(num);
                    fdb.insertIncoming(in);
                }
                fdb.close();
            }
        }
    }
```

```
private void stopCall() {            //运用反射原理调用被隐藏的 stopCall()函数
    try {
      Method method;
      method=Class.forName("android.os.ServiceManager").getMethod(
              "getService",String.class);
      IBinder binder=(IBinder) method.invoke(null,
              new Object[] { Context.TELEPHONY_SERVICE });
      ITelephony telephony=ITelephony.Stub.asInterface(binder);
      telephony.endCall();
      telephony.cancelMissedCallsNotification();
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
```

通过以上两个例子可以清楚地发现反射机制在编写 Java 程序和 Android 应用时的两个重要作用：获取不知名的类和使用一些已被隐藏的类。下面将对反射机制的 API 和内部函数及实现原理进行详细的分析。

4.3 反射机制 API 分析

Java 反射机制让程序员在程序运行状态中，可以对于任意一个类，都能够获取这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；总的来说反射机制主要提供了以下 4 种功能：①在运行时判断任意一个对象所属的类；②在运行时构造任意一个类的对象；③在运行时判断任意一个类所具有的成员变量和方法；④在运行时调用任意一个对象的方法并生成动态代理。

4.3.1 反射机制 API 分析概述

在 Java 核心库中主要有以下几个类承担了类反射机制的全部功能，它们分别是：Class、Field、Constructor、Method、Array，这几个类的功能以及它们所提供的方法见表 4.1。

表 4.1　反射机制中的关键类

类名	简　介	关 键 方 法
Class	Class 类的实例表示正在运行的 Java 应用程序中的类和接口	forName()：返回与带有给定字符串名的类或接口相关联的 Class 对象
		getFields()：获得类的 public 类型的属性
		getDeclaredFields()：获得类的所有属性
		getMethods()：获得类的 public 类型的方法
		getDeclaredMethods()：获得类的所有方法
		getMethod(String name,Class[]parameterTypes)：获得类的特定方法，name 参数指定方法的名字，parameterTypes 参数指定方法的参数类型
		getConstructors()：获得类的 public 类型的构造方法
		getConstructor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes 参数指定构造方法的参数类型
		newInstance()：通过类的不带参数的构造方法创建这个类的一个对象
Field	Field 提供有关类或接口的单个字段的信息，以及对它的动态访问权限。反射的字段可能是一个类(静态)字段或实例字段	get()：返回指定对象上此 Field 表示的字段的值
		getDeclaringClass()：返回表示类或接口的 Class 对象，该类或接口声明由此 Field 对象表示的字段
		getModifiers()：以整数形式返回由此 Field 对象表示的字段的 Java 语言修饰符
		set()：将指定对象变量上此 Field 对象表示的字段设置为指定的新值
		toString()：返回一个描述此 Field 的字符串
Method	Method 提供关于类或接口上单独某个方法(以及如何访问该方法)的信息。所反映的方法可能是类方法或实例方法(包括抽象方法)	getDeclaringClass()：返回表示声明由此 Method 对象表示的方法的类或接口的 Class 对象
		invoke()：对带有指定参数的指定对象调用由此 Method 对象表示的底层方法
		getModifiers()：以整数形式返回此 Method 对象所表示方法的 Java 语言修饰符
		getName()：以 String 形式返回此 Method 对象表示的方法名称
		getReturnType()：返回一个 Class 对象，该对象描述了此 Method 对象所表示的方法的正式返回类型
Constructor	Constructor 提供关于类的单个构造方法的信息以及对它的访问权限	newInstance()：使用此 Constructor 对象表示的构造方法来创建该构造方法的声明类的新实例，并用指定的初始化参数初始化该实例
		isVarArgs()：如果声明此构造方法可以带可变数量的参数，则返回 true；否则返回 false
		getDeclaringClass()：返回 Class 对象，该对象表示声明由此 Constructor 对象表示的构造方法的类
		getModifiers()：以整数形式返回此 Constructor 对象所表示构造方法的 Java 语言修饰符
		toString()：返回描述此 Constructor 的字符串
Array	Array 类提供了动态创建和访问 Java 数组的方法	newInstance()：创建一个具有指定的组件类型和长度的新数组
		set()：将指定数组对象中索引组件的值设置为指定的新值
		get()：返回指定数组对象中索引组件的值
		getLength()：以 int 形式返回指定数组对象的长度

从表 4.1 中可以很清晰地看出，反射机制所涉及的各个类以及其相关方法，主要实现了对三类信息的获取和使用：构造函数、字段和方法。而在这些类中，最重要的类是 Class 类，该类为前面提到的每一类信息设计封装了 4 种独立的反射调用，可以根据不同的需要调用相应的反射方法获取了相应所需的反射类型的实例对象。

由此可以看出 Class 类的重要性——要想得到 Field、Method 以及 Constructor 等类的实例对象必须经过 Class 类，因而可以得出 Class 类是反射机制的源头。

4.3.2 代理模式 API 分析

代理模式是常用的 Java 设计模式，它的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类可以提供对另一个对象的访问，同时隐藏实际对象的具体实例。代理一般会实现它所表示的实际对象的接口。代理可以访问实际对象，但是延迟实现实际对象的部分功能，实际对象实现系统的实际功能，代理对象对客户隐藏了实际对象。客户不知道它是与代理打交道还是与实际对象打交道。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。而动态代理则是在程序运行时，运用反射机制动态创建而成。

因此，这种机制既然是依托于反射机制而建立，那么本报告也对动态代理机制进行一定程度的分析，代理类 Proxy 的相关信息如表 4.2 所示。

表 4.2 代理类 Proxy 的相关信息

类名	简　介	相 关 方 法
Proxy	Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类	构造方法
		Proxy()：使用其调用处理程序的指定值从子类(通常为动态代理类)构建新的 Proxy 实例
		关键方法
		getInvocationHandler()：返回指定代理实例的调用处理程序
		getProxyClass()：返回代理类的 java.lang.Class 对象，并向其提供类加载器和接口数组
		isProxyClass()：当且仅当指定的类通过 getProxyClass 方法或 newProxyInstance 方法动态生成为代理类时，返回 true
		newProxyInstance()：返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序

点拨 从 Java API 这一层次程序员不能很好地发现动态代理机制和类反射机制有何种联系，实际上它们的具体联系是：它们都使用同一种思路完成它们各自的功能——都是将方法的具体实现放在虚拟机中，而且共享部分关键的实现函数和数据结构。

4.3.3 元数据注释机制 API 分析

在 reflect 文件夹中除了反射机制和动态代理之外，还包含一个 Annotation.cpp 文件，

它就是注释类，是 Java5 的一个新特性，JDK5 引入了 Metedata(元数据)的概念，很容易地就能够调用 Annotations 提供一些本来不属于程序的数据。所谓的元数据，就是“关于数据的数据”。用于标示程序中各个数据、方法以及属性的信息，还包括对构造器的声明、域声明、局部变量声明以及方法声明等重要的功能。因此，适当地应用元数据可以很好地提高程序的功能性和稳定性。在 Class 类中就为元数据封装了相应的方法，在取得相应的注释类型实例后，再通过反射机制，可以对元数据进行取得或修改。“注释”类的相关信息如表 4.3 所示。

表 4.3　注释类的相关信息

类名	简　介	相 关 方 法
Annotation	用于标示程序中各个数据、方法以及属性的信息，还包括对构造器的声明、域声明、局部变量声明以及方法声明等重要的功能	annotationType()：返回此 annotation 的注释类型
		equals()：如果指定的对象表示在逻辑上等效于此接口的注释，则返回 true
		hashCode()：返回此 annotation 的哈希码
		toString()：返回此 annotation 的字符串表示形式

点拨　反射机制虽然是以 Java API 的形式服务于上层的 Java 应用，但它和 Dalvik 虚拟机的关系是十分紧密的。实际上，反射机制的 API 的具体实现是依托于虚拟机 Dalvik，它的实现机理简单来说是：上层 Java 方法通过 JNI 机制(本地调用机制)调用 Dalvik 虚拟机内部函数以完成其具体功能，再将运行结果逐层返回给上层应用。也就是说反射机制的执行模块是在虚拟机中，脱离虚拟机独立存在的反射机制只能是一个空架子。

4.4　反射机制的“三层”实现体系

4.4.1　类反射机制在 Dalvik 虚拟机内部的实现

Java API 是面向上层应用开发，而这些封装在 Java 核心库中的 API 所涉及的各个方法的具体实现是在 Dalvik 虚拟机中，而中间起到过渡作用的是 JNI 本地方法调用机制。在本节中将详细介绍一下 Java API 在虚拟机内部实现的机理，首先通过观察图 4.1 可以很好地对反射机制实现所需要的三层结构有一个直观的认识。

在图 4.1 中位于最上层的是 Java API，直接决定了虚拟机该如何工作；中间层是 JNI 机制，用于衔接 Java 方法和底层的本地方法；下层是本地方法，用于接口方法的直接实现。报告将分别对类反射机制的 5 个关键类进行详细分析，着重分析其工作流程以及其关键的实现函数分析。

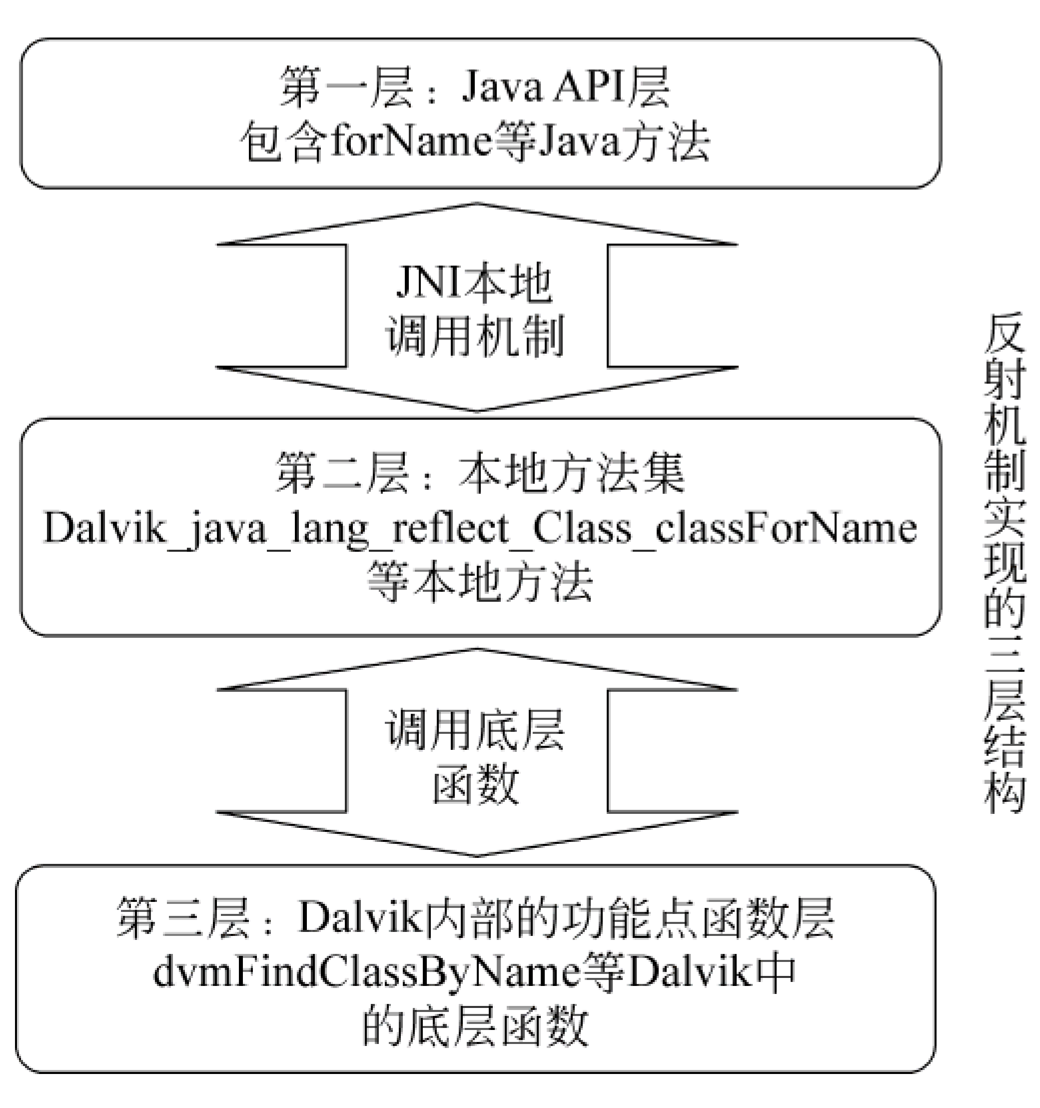


图 4.1　类反射机制实现的三层结构图

4.4.2 三层结构实例展示

在函数详细分析之前，先通过一段具体代码直观展示一下类反射机制的三层调用关系，提前熟悉这三层调用关系可以很好地帮助读者对后面内容的理解。

下面以 Class 类中的静态方法 Class.forName 为例，介绍一下这个实现流程。

先来看反射机制的第一层——Java API 层：在核心库中找到 Class 类的定义处 Java\lang\Class.java，从这个路径可以看出 lang 文件夹就是在 Java 编程中默认引入的 lang 包，在这个 Class.java 文件中找到 Class.forName 方法的原始定义处。

代码清单 4.3 libcore\luni\src\main\java\java\lang\class.java：forName()源代码

```
public static Class<?> forName(String className,boolean initializeBoolean,
            ClassLoader classLoader) throws ClassNotFoundException {
        if (classLoader==null) {
            classLoader=ClassLoader.getSystemClassLoader();
        }
        Class<?> result;
        try {
            result=classForName(className,initializeBoolean,
                    classLoader);
        } catch (ClassNotFoundException e) {
            Throwable cause=e.getCause();
            if (cause instanceof ExceptionInInitializerError) {
                throw (ExceptionInInitializerError) cause;
            }
            throw e;
        }
        return result;
    }
```

通过简单的分析，这个方法实际上是通过调用另外一个 classForName 方法根据一个已知的类名从虚拟机中加载并初始化这个类，在源码中再次回溯找到这个 classForName 方法的定义。

代码清单 4.4 libcore\luni\src\main\java\java\lang\class.java：classForName 源代码

```
    static native Class<?> classForName(String className, boolean initialize Boolean, ClassLoader
classLoader) throws ClassNotFoundException;
```

然而这个方法并没有具体的方法体，需要注意到这个方法实际上是一个 Native 方法，这就意味着 Class.forName 方法在这里需要通过 JNI 机制调用虚拟机中的本地方法去代它完成既定的目标，回溯到 android_dalvik_source\vm\native 文件夹(这就是第二层——本地方法集)，其中 java_lang_Class.cpp 文件便是对应 Class.java 文件中所涉及的所有本地方法，在 java_lang_Class.cpp 文件中找到该方法的 C 语言实现。

代码清单 4.5 dalvik\vm\native\ java_lang_Class.cpp：Dalvik_java_lang_Class_classForName

源代码

```
static void Dalvik_java_lang_Class_classForName(const u4* args,JValue* pResult)
{
    StringObject* nameObj= (StringObject* ) args[0];
    bool initialize= (args[1] != 0);
    Object* loader= (Object* ) args[2];

    RETURN_PTR(dvmFindClassByName(nameObj,loader,initialize));
}
```

对这段代码简单分析可以发现，这个函数实际上是通过调用 dvmFindClassByName 函数实现其功能的，而 dvmFindClassByName 函数的具体定义在 android_dalvik_source\vm\native\InternalNative. cpp 文件中(这就是第三层——Dalvik 虚拟机内部的功能点函数层)。

代码清单 4. 6　libcore\luni\src\main\java\java\lang\class. java：dvmFindClassByName 源代码

```
ClassObject* dvmFindClassByName(StringObject* nameObj,Object* loader,
    bool doInit)
{
    ClassObject* clazz=NULL;
    char* name=NULL;
    char* descriptor=NULL;

    if (nameObj==NULL) {
        dvmThrowNullPointerException("name==null");
        goto bail;
    }
    name=dvmCreateCstrFromString(nameObj);
```

在这个函数定义中，可以发现 dvmFindClassByName 函数还调用了其他一些函数完成其功能。Class. forName 的实现过程很好地验证了确实存在一种“三层关系”——核心库、JNI 以及虚拟机，即核心库中的 Java 方法通过 JNI 机制调用虚拟机源码中 vm 文件夹下 native 文件夹中的本地方法集，再由这些本地方法函数调用虚拟机中另外的一些功能点函数去完成相应的目标。

点拨　虚拟机在实现一个 API 的功能时的执行流程是：上层的接口负责接收参数、指令，经过中间 JNI 机制的转发将这些控制信息交付给底层虚拟机内部的执行函数实现其功能，最后将结果逐层返回给上层调用，完成对一个方法的实现。

4.5　反射机制实现分析

4.5.1　Class 类详细分析

对于 Class 类，读者已经知道它是反射机制的源头，而它究竟能做些什么以及它是如何实现这些功能的，本节将对 Class 类进行详细分析。

1. 构造函数的处理

根据需求的不同可分为以下 4 个方法。

Constructor getConstructor()——获得使用特殊的参数类型的公共构造函数。

Constructor[]getConstructors()——获得类的所有公共构造函数。

Constructor getDeclaredConstructor()——获得使用特定参数类型的构造函数。

Constructor[]getDeclaredConstructors()——获得类的所有构造函数。

首先通过观察 getConstructor()函数代码绘制流程图如图 4.2 所示。

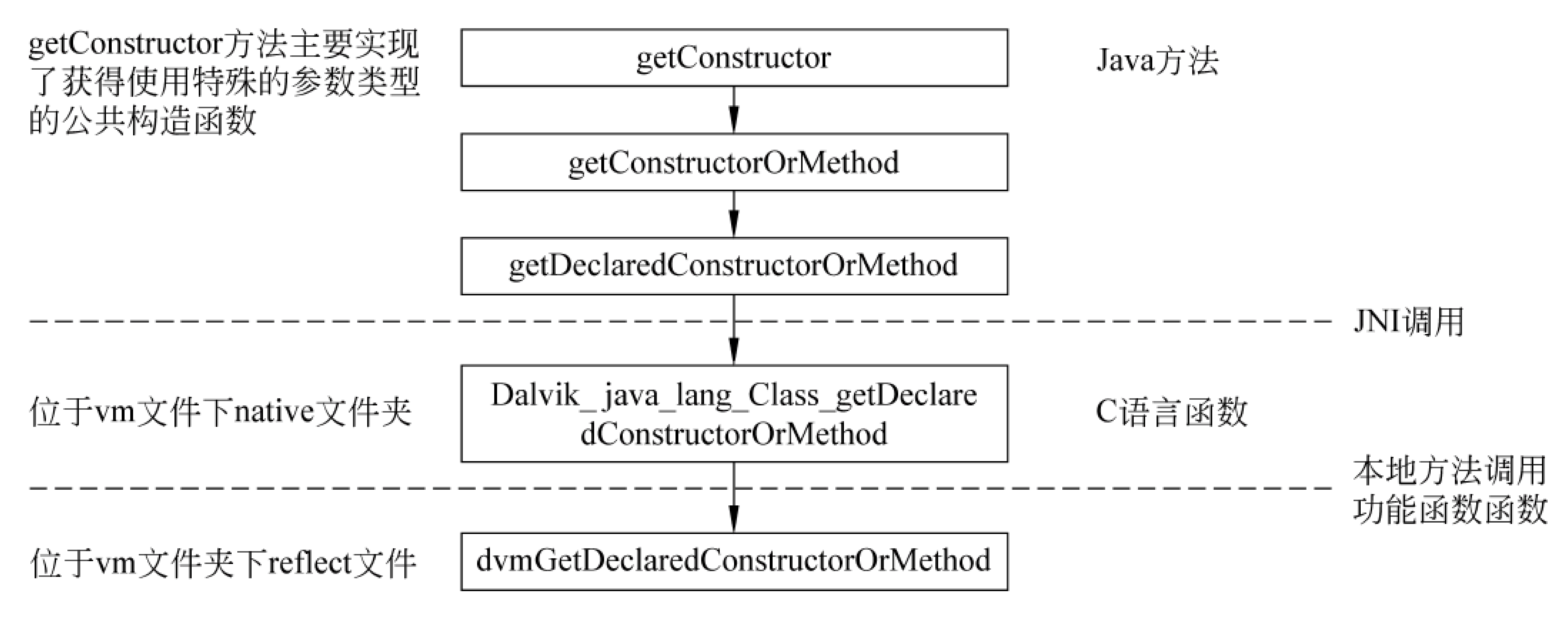


图 4.2 getConstructor()方法流程图

再绘制 getConstructors()方法的流程图如图 4.3 所示。

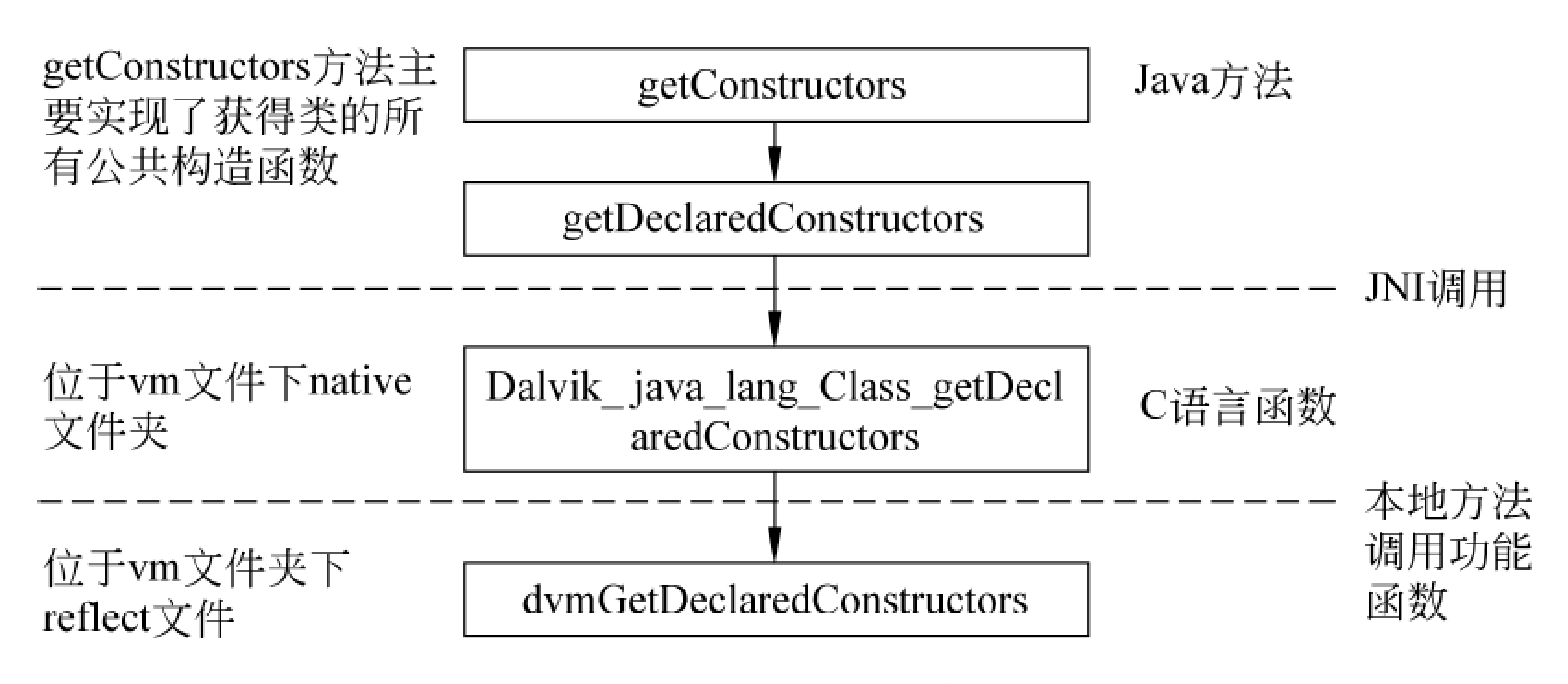


图 4.3 getConstructors()方法流程图

从图 4.3 中经分析可知，getConstructors()方法实际上是通过调用本地方法 getDeclaredConstructorOrMethod 接口从 Java 语言过渡到 C 语言，再通过虚拟机内部本地方法集 Dalvik_java_lang_Class_getDeclaredConstructorOrMethod 将相关参数传递给实际执行函数 dvmGetDeclaredConstructorOrMethod 完成 getConstructor()方法的实际功能；在对 getDeclaredConstructor()方法进行分析时，发现 dvmGetDeclaredConstructors 函数是该方法的本地实现函数。同时经过对源码的验证，发现该 dvmGetDeclaredConstructorOrMethod 函数和 dvmGetDeclaredConstructors 函数确实封装在 reflect.cpp 中，由此可见上文的分析思路与结果是正确的，也证明了反射机制的具体实现确实在虚拟机内部。另外，在报告的“关键函数详细分析”部分中，本文会对以上两个 dvmGetDeclaredConstructorOrMethod 函数和

dvmGetDeclaredConstructors 函数的内部实现机制进行具体的分析。

利用这种方法本文为其他两个方法 getDeclaredConstructor()以及 getDeclaredConstructor()均绘制了方法流程图并对它们在实现过程中所调用的函数进行了归纳总结,发现其对应关系如图 4.4 所示。

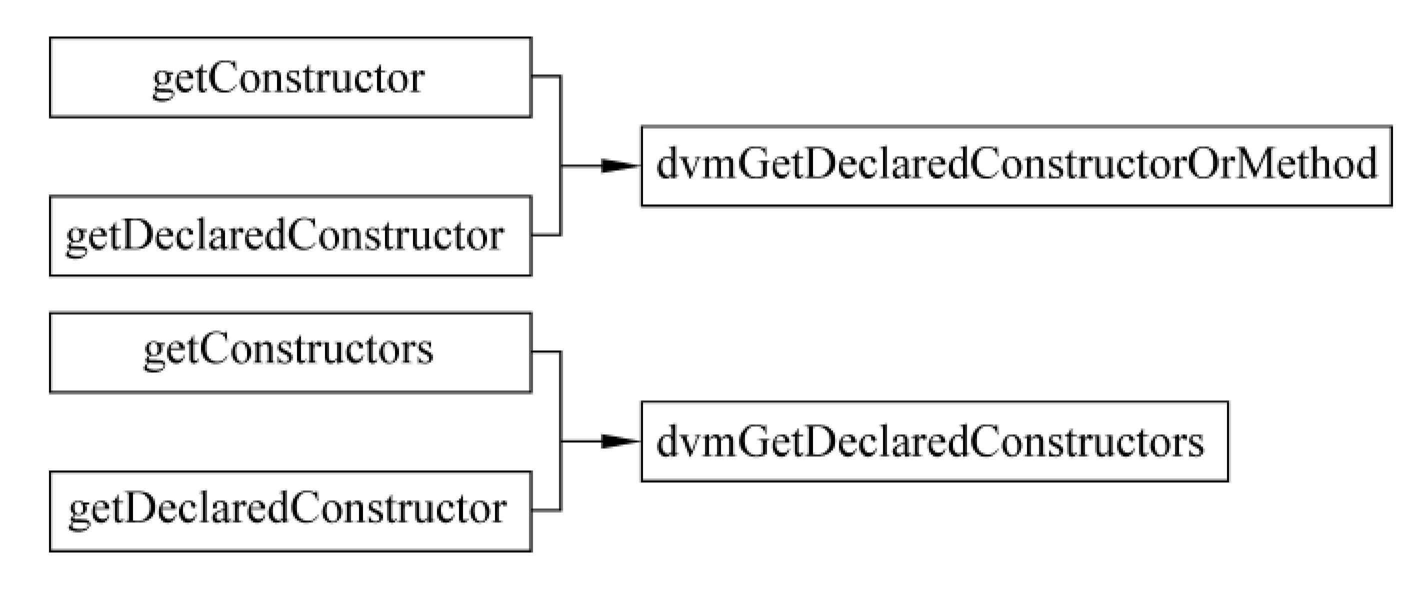


图 4.4　构造函数相关的对应关系图

点拨　这 4 个 Java 方法对应底层两个具体实现函数,对底层函数的分析发现,通过区分底层函数的入口参数就可以完成不同 Java 方法的实现目的,也就实现了对函数的高效利用。

2. 对方法的处理

根据需求的不同可分为以下 4 个方法。

Method getMethod()——使用特定的参数类型,获得命名的公共方法。

Method[]getMethods()——获得类的所有公共方法。

Method getDeclaredMethod()——获得类声明的命名的方法。

Method[]getDeclaredMethods()——获得类声明的所有方法。

首先通过观察 getMethod()方法的代码绘制流程图如图 4.5 所示。

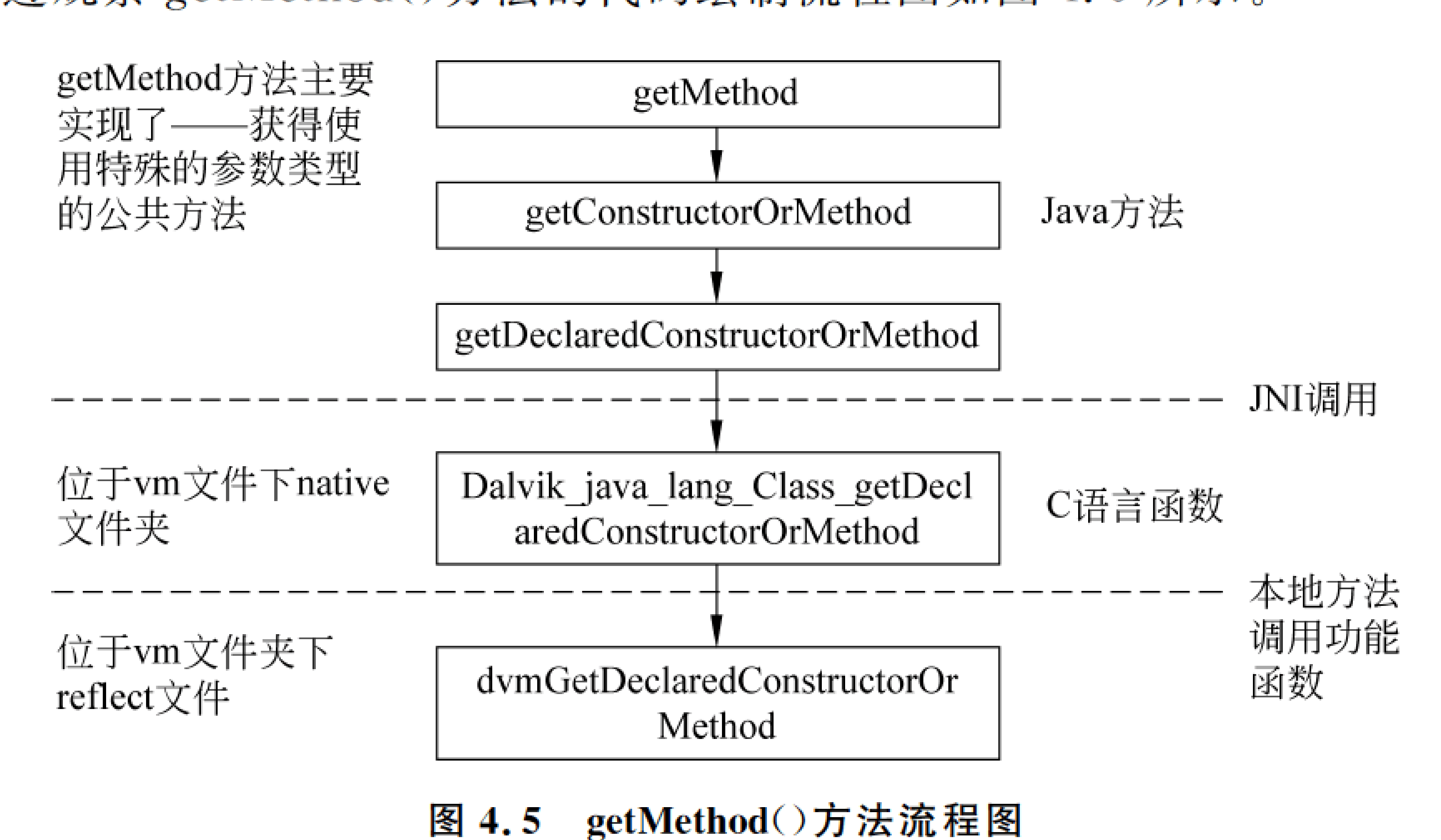


图 4.5　getMethod()方法流程图

再绘制 getMethods()方法的流程图如图 4.6 所示。

获取类中方法的这组接口的实现原理与结构和构造函数一样,都是通过调用本地方法完成其方法功能。通过对另外两个方法 getDeclaredMethod()和 getDeclaredMethods()绘制流程图并归纳总结发现上层接口与底层具体实现函数有如下对应关系,如图 4.7 所示。

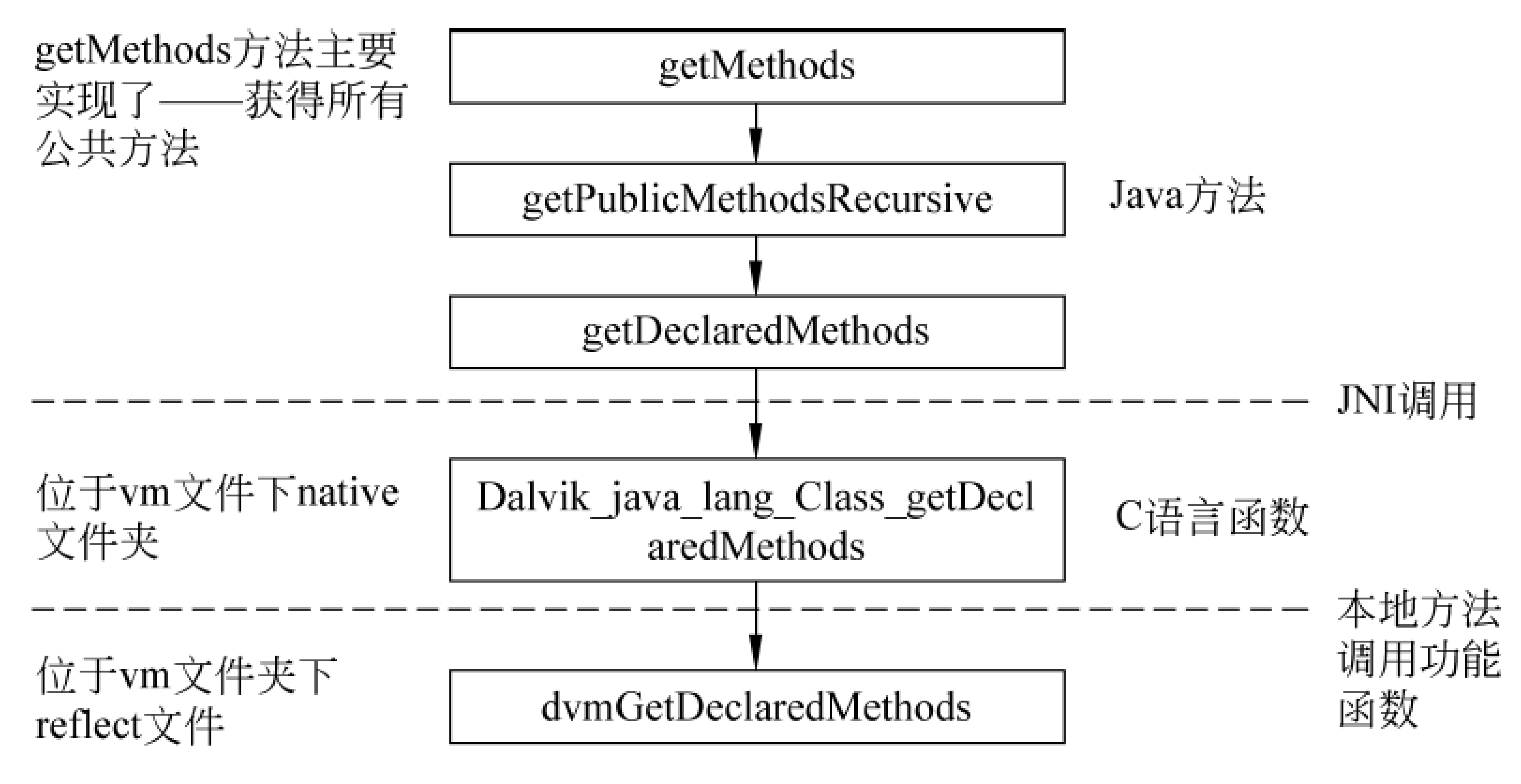


图 4.6　getMethods()方法的流程图

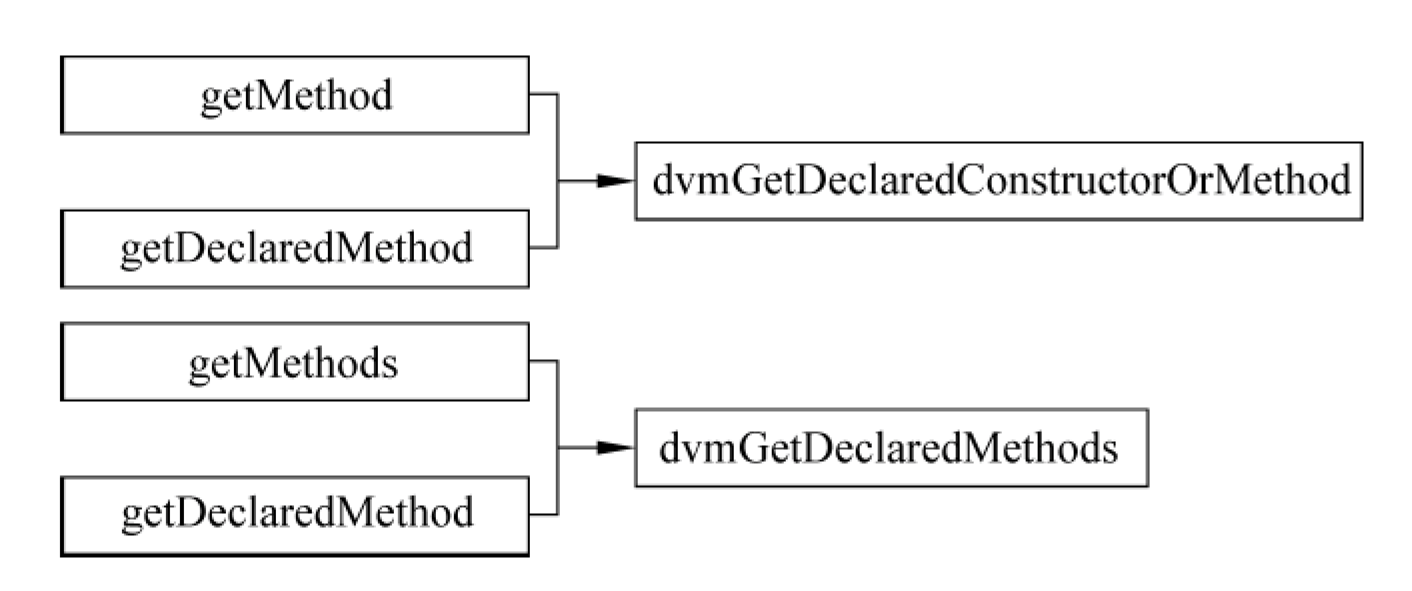


图 4.7　方法相关的对应关系图

从图 4.7 可以知道，Class 类中用于获得方法信息的 4 个反射方法分别对应底层的两个实现函数，和获取构造函数信息的情形相同，这里涉及的两个底层执行函数会在报告的“关键函数详细分析”部分中进行详细分析。

3. 对字段的处理

根据需求的不同可分为以下 4 个方法。

Field getField()——获得命名的公共字段。

Field[]getFields()——获得类的所有公共字段。

Field getDeclaredField()——获得类声明的命名的字段。

Field[]getDeclaredFields()——获得类声明的所有字段。

观察代码 getField()方法实现结构图，如图 4.8 所示。

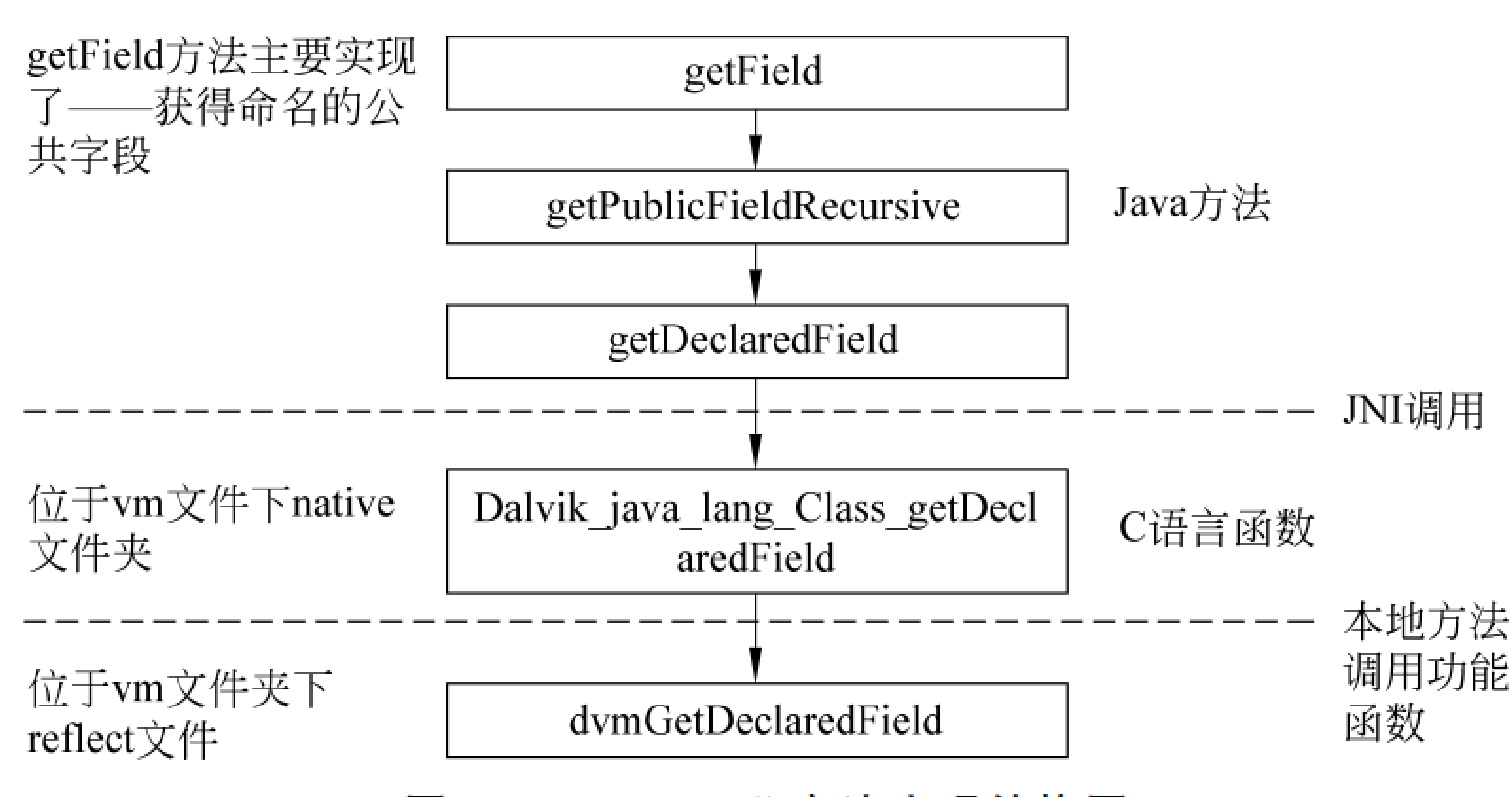


图 4.8　getField()方法实现结构图

getFields()方法的实现结构图，如图 4.9 所示。

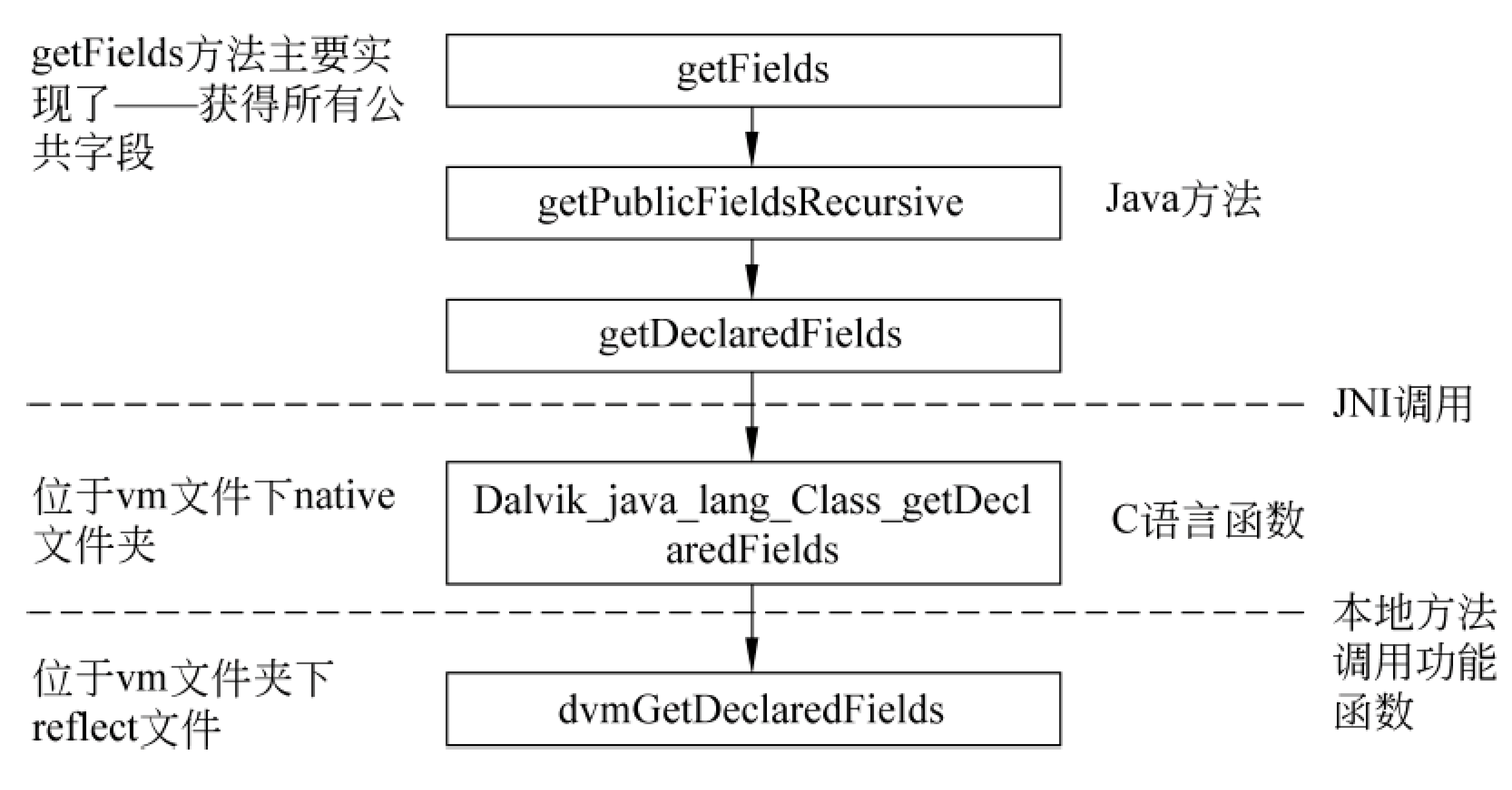


图 4.9　getFields()方法的实现结构图

获取类中字段信息的这组接口的实现原理与结构和构造函数一样，都是通过调用本地方法完成其方法功能。通过对另外两个方法 getDeclaredField()和 getDeclaredFields()绘制流程图并归纳总结发现上层接口与底层具体实现函数有如下对应关系，如图 4.10 所示。

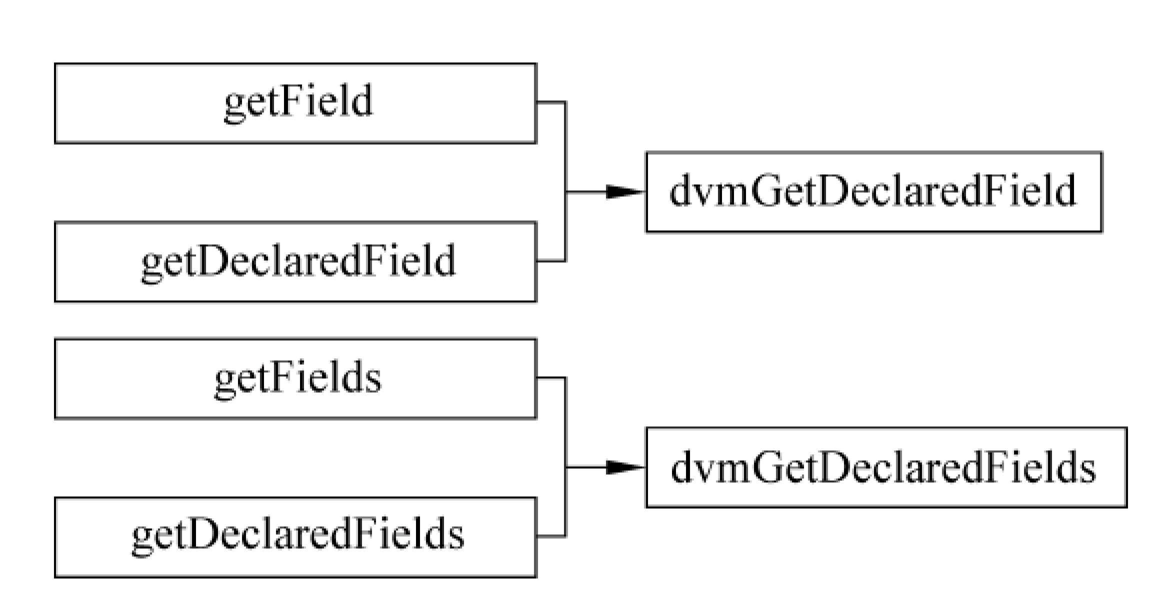


图 4.10　字段相关的对应关系图

从图 4.10 可以知道，Class 类中用于获得字段信息的 4 个反射方法分别对应底层的两个实现函数，和获取构造函数信息的情形相同，这里涉及的两个底层执行函数会在报告的“关键函数详细分析”部分中进行详细分析。

至此，本节完成了对 Class 类中所涉及的三组 12 个反射方法的实现原理以及实现流程架构的分析。

4.5.2　Constructor 类详细分析

Constructor 类提供关于类的单个构造方法的信息以及对它的访问权限。其中最关键的方法是 newInstance()，用于调用相关联的构造函数实例化一个类对象。报告主要对这个方法的实现机理进行分析并展示其实现流程。

首先通过该方法的实现结构图了解一下它的实现机理，如图 4.11 所示。

从图 4.11 中可以看到这个 newInstance()方法实际上在取得入口参数之后立即返回调用本地接口 constructNative，由此过渡到虚拟机内部执行，在虚拟机内部可以发现本地方法随后调用了 dvmInvokeMethod 函数去执行该构造方法。至此，完成了实例化一个对象的工作。这里需要指出的是，dvmInvokeMethod 函数实际是属于解释器范畴，本书就不再详述。

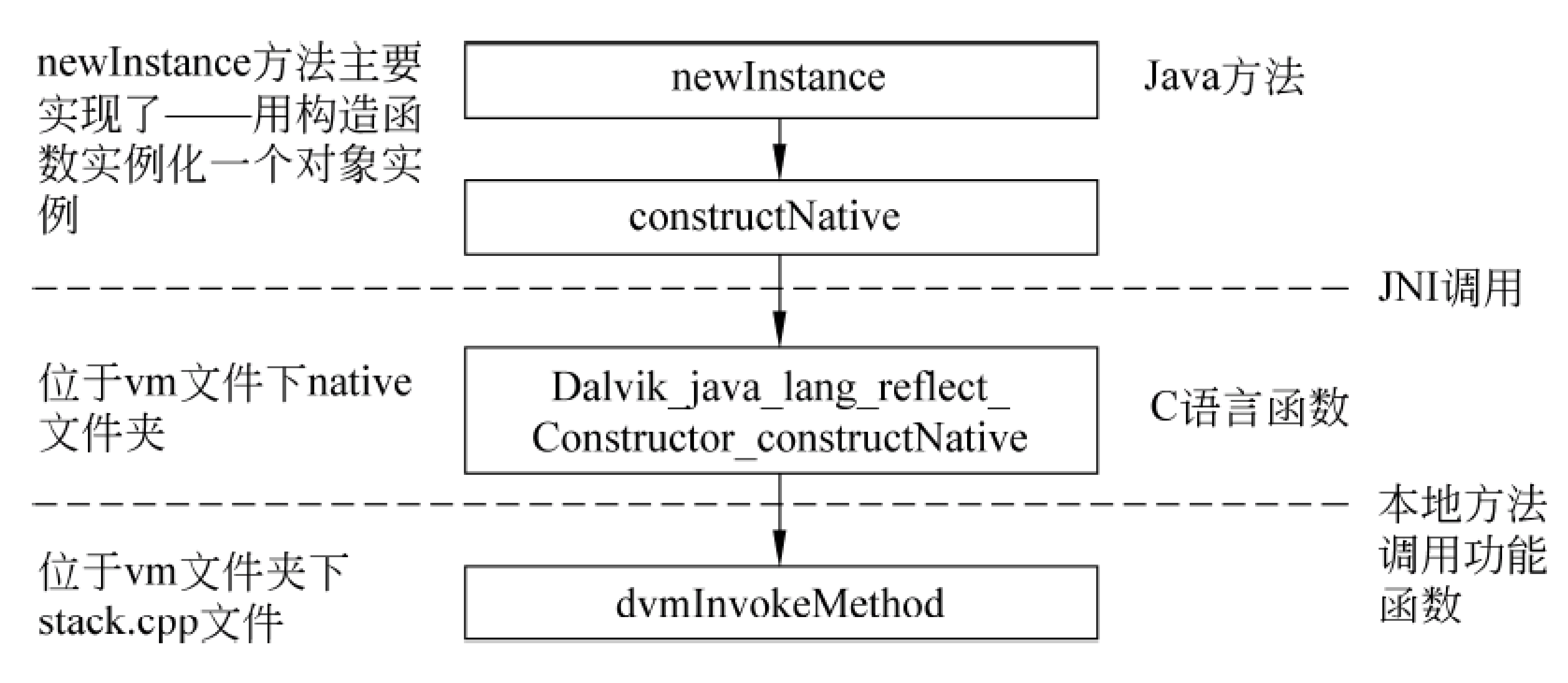


图 4.11　newInstance 方法实现结构图

4.5.3　Method 类详细分析

Method 类提供关于类或接口上单独某个方法的信息。所反映的方法可能是静态方法或虚拟方法，它是用来封装反射类方法的一个类。此类中最重要的方法为 invoke，它的主要作用是通过反射机制激活一个静态或虚拟方法。此函数具体的实现结构如图 4.12 所示。

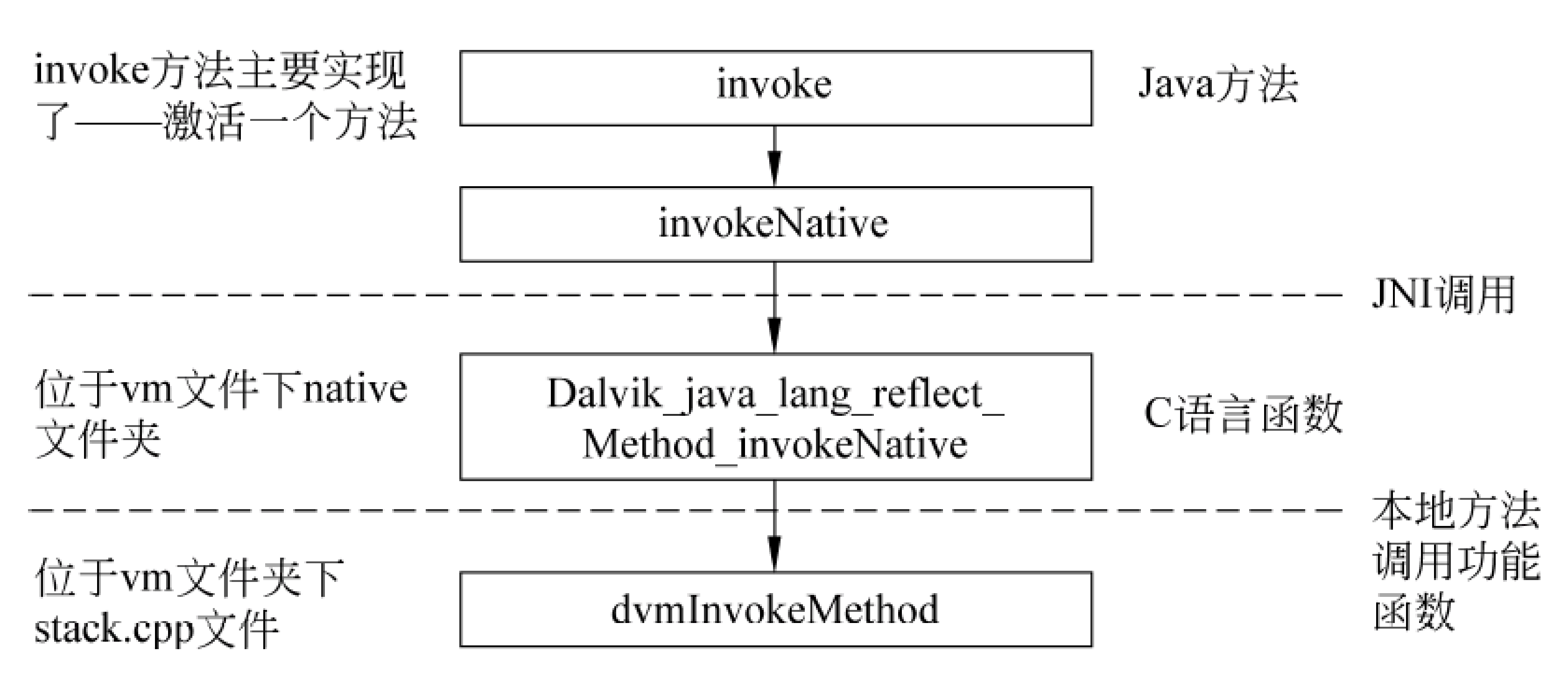


图 4.12　invoke 方法实现结构图

结合代码和结构图可以得知，invoke 方法首先需要取得入口参数，然后调用本地接口 invokeNative，运用 JNI 机制将 Java 方法转换为 C 语言函数，将该方法转换到虚拟机内部执行。本地方法 Dalvik _ java _ lang _ reflect _ Method _ invokeNative 随后调用了 dvmInvokeMethod 函数将方法激活。综上所述，invoke()方法就是这样通过类反射机制将一个静态或虚拟的方法激活的。

4.5.4　Field 类详细分析

Field 类提供有关类或接口的属性的信息，以及对它的动态访问权限。反射的字段可能是一个静态或动态字段，简单的理解可以把它看成一个封装反射类的属性的类。此类中相对重要的方法是 get()方法以及 set()方法，下面分析一下这两个方法的具体实现过程。

首先本文先分析 get()方法，此方法主要的功能是获得目标字段的值并且进行封装，此函数的具体流程如图 4.13 所示。

结合代码和流程图可以看出，get()方法获得入口参数之后直接调用本地接口 getField，运用 JNI 机制将上层的 Java 方法转换为 C 语言函数，将该方法转换到虚拟机内部执行。本地方法 Dalvik_java_lang_reflect_Field_getField 的主要功能是将一个原始字段进行封装并

且获得其中的值,在这里只简单描述该函数的流程,具体分析将在后文中的函数详细分析部分进行介绍。本函数的入口参数 args 为一个常量,pResult 为虚拟机参数。该函数首先定义了所需要的相关变量,接下来函数并没有直接获得字段中的值,而是先调用 validateFieldAccess 函数,来验证字段是否可访问并且返回一个指向结构体的字段 field。在获得函数 validateFieldAccess 返回的字段指针 field 之后,主函数对该指针进行非空判断,如果 field 不为空,则函数调用 getFieldValue 函数。

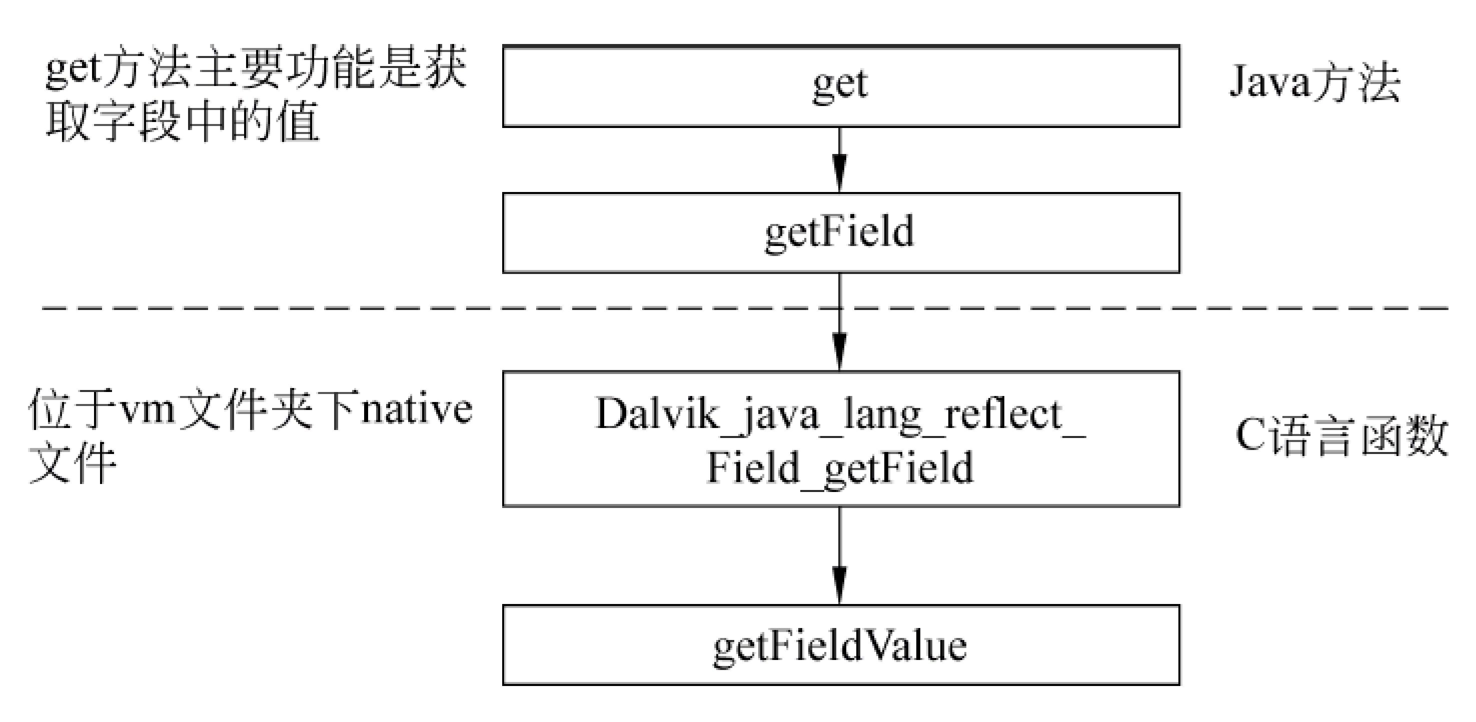


图 4.13　get()方法实现流程图

getFieldValue 函数虽然很简短,但它却是实现提取字段值的关键函数,同样也是 get()方法实现的核心函数,结合代码能更清晰地了解此函数的工作流程。getFieldValue 函数首先判断该字段是静态字段还是实例字段,然后通过调用 getStaticFieldValue 和 getInstFieldValue 函数分别将静态字段和实例字段的值提取出来。在 getFieldValue 函数结束后本地函数 Dalvik_java_lang_reflect_Field_getField 调用 dvmBoxPrimitive 函数将字段进行封装,然后赋给 result。调用 dvmReleaseTrackedAlloc 函数,停止跟踪这个对象。返回刚刚获得的封装好的数据对象 result。到此,get()方法就完整地实现了提取字段值的功能。

接下来分析 set()方法,此方法主要的功能是从一个已经被封装好的字段中提取值,此函数的具体流程如图 4.14 所示。

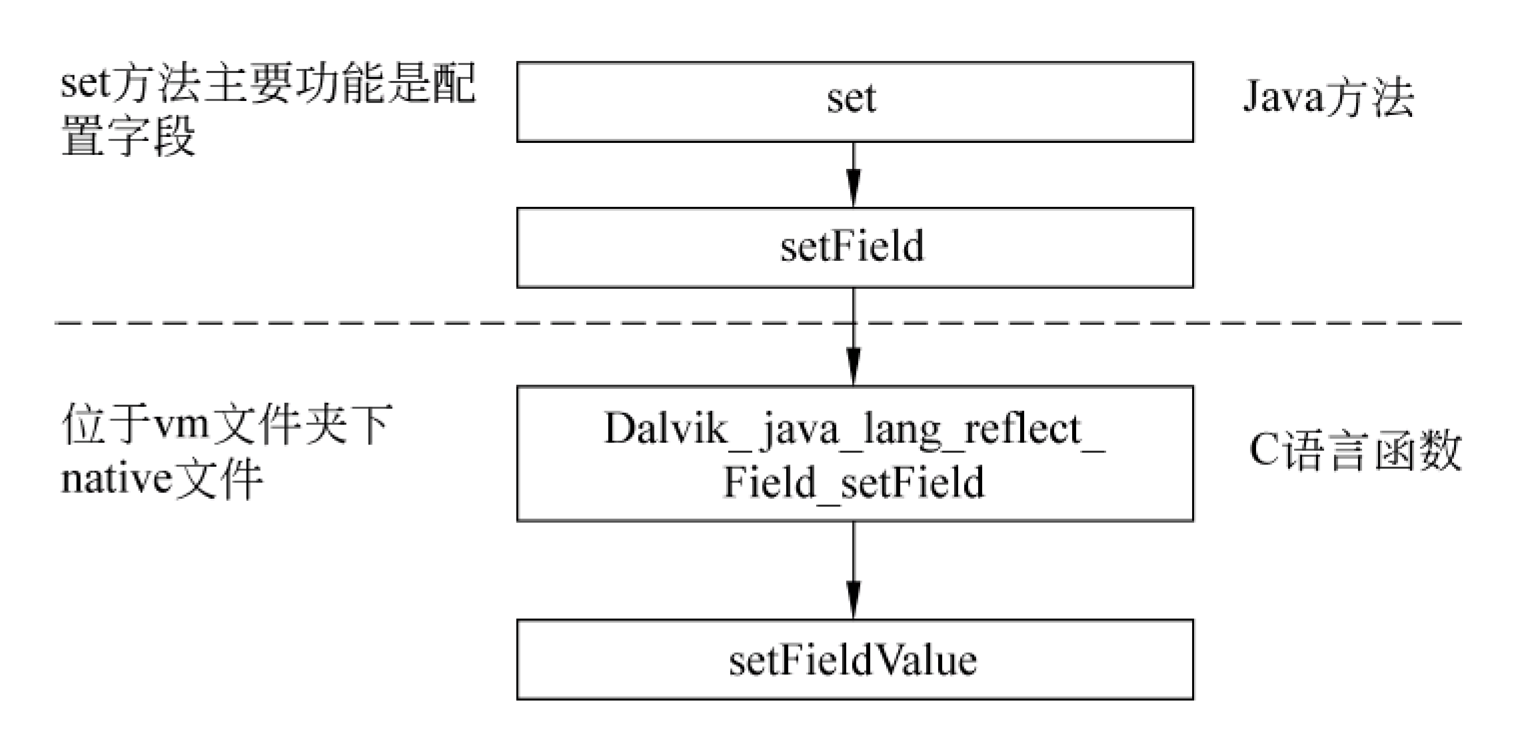


图 4.14　set()方法实现流程图

结合代码和流程图可以看出,set()方法获得入口参数之后直接调用本地借口 setField,运用 JNI 机制将上层的 Java 方法转换为 C 语言函数,将该方法转换到虚拟机内部执行。本地方法 Dalvik_java_lang_reflect_Field_setField 的主要功能是获得一个封装好的原始字段中的值,在这里只简单描述该函数的流程,具体分析将在后文中的函数详细分析部分进行介

绍。此函数入口参数 args 为一个常量，pResult 为虚拟机参数。该函数首先定义了所需要的相关变量，随后调用 dvmUnboxPrimitive 函数，这个函数的功能是打开一个已经封装的原始类型，具体函数与本方法关系不大，在这里就不做介绍了。

在打开封装之后，函数调用 validateFieldAccess 函数进行访问验证，获得一个字段指针 field。对 field 进行非空判断，如果不为空则调用 setFieldValue 函数。setFieldValue 函数首先判断该字段是静态字段还是实例字段，然后调用 setStaticFieldValue 和 setInstFieldValue 函数将值配置到相应类型的字段中。setFieldValue 函数是 set()方法中较为核心的部分。

至此，set()方法完成了它的主要功能，成功地打开了一个封装好的原始字段并配置相关字段。

4.5.5 反射机制对 Proxy 类和 Annotation 类功能上的支持

首先，已经明确动态代理和元数据注释两种机制是建立在反射机制的实现原理之上，因此本书主要围绕两种机制在虚拟机内部的具体实现进行介绍，不对上层的 API 进行分析。

本书首先对两种机制中的各个接口进行了分析，归纳分析了其中的各个本地方法接口，得到的结果是：这两种机制也是遵循“三个层次”这一实现结构。但两种机制的底层实现函数并没有封装在虚拟机源码中的 reflect. cpp 中，而是分别存在于 Proxy. cpp 和 Annotation. cpp 文件中，这三个文件统一存放在虚拟机源码的 reflect 文件夹下，因此，再次印证了本文之前的分析——即这两种机制确实是依托于反射机制而设计的，通过对源码的简单分析发现这两种机制在对相关类信息的处理上，所用到的方法思路和反射机制非常相似。

4.5.6 核心函数详细分析

1. Dalvik_java_lang_reflect_Method_invokeNative 函数详细分析

函数流程图如图 4.15 所示。

本函数是 Method 类封装的 invoke 方法的具体实现，它的重要功能是在通过反射机制获取一个方法后去激活使用这个方法，使 Java 程序的灵活性大大提高。本函数的入口参数比较多，因此，主要介绍一下几个关键的参数以及中间变量。首先是 Object * methObj——目标对象、ClassObject * declaringClass——目标方法所属的类对象、int slot——方法在类方法表中的序号以及 Method * meth——方法指针。

主函数首先调用 dvmSlotToMethod 函数根据 slot 变量获取该目标方法在类对象方法区中的位置，并将这个方法返回给方法指针 meth。至此，此函数完成取到方法的工作。

主函数随后便将对这个方法进行一系列判断，首先是判断该方法是否为静态方法，如果是，则判断相应的类是否被加载且被初始化，在初始化后将该方法的指针 meth 直接交付给 dvmInvokeMethod 函数执行，否则将首先对所属类进行进一步判断，判断其是否为一个接口，如是则加载并初始化这个接口，如果该所属类不是一个接口，则进行下一步骤；判断 methObj——目标对象是否为预期的所属类的一个实例对象，随后，主函数将会查找这个方法的准确入口，并将正确的方法入口赋值给 meth 指针，最后将该方法的指针 meth 直接交付给 dvmInvokeMethod 函数执行。主函数到这已经取得了正确的方法指针 meth，因此，将

调用 dvmInvokeMethod 函数对取得的方法解释执行，并将结果返回给 result。

至此，完成了对本地方法 Dalvik_java_lang_reflect_Method_invokeNative 函数的分析。

开始
获得入口参数 const u4* args JValue* pResult
调用 dvmSlotToMethod 函数获取方法位置
调用 dvmIsStaticMethod 判断该方法是否为静态方法
是
调用 dvmInitClass 函数对该类进行初始化
否
调用 dvmLsInterfaceClass 函数判断对象是否为一个预期的实例类
是
调用 dvmInitClass 函数对该类进行初始化
否
调用 dvmVerifyObjectInClass 函数判断对象是否为一个预期的实例类
调用 dvmGetVirtualized Method函数查找该方法
调用 dvmInvokeMethod 完成对方法的激活
结束

图 4.15　Dalvik_java_lang_reflect_Method_invokeNative 函数流程图

2. Dalvik_java_lang_reflect_Proxy_generateProxy 详细分析

函数流程图如图 4.16 所示。

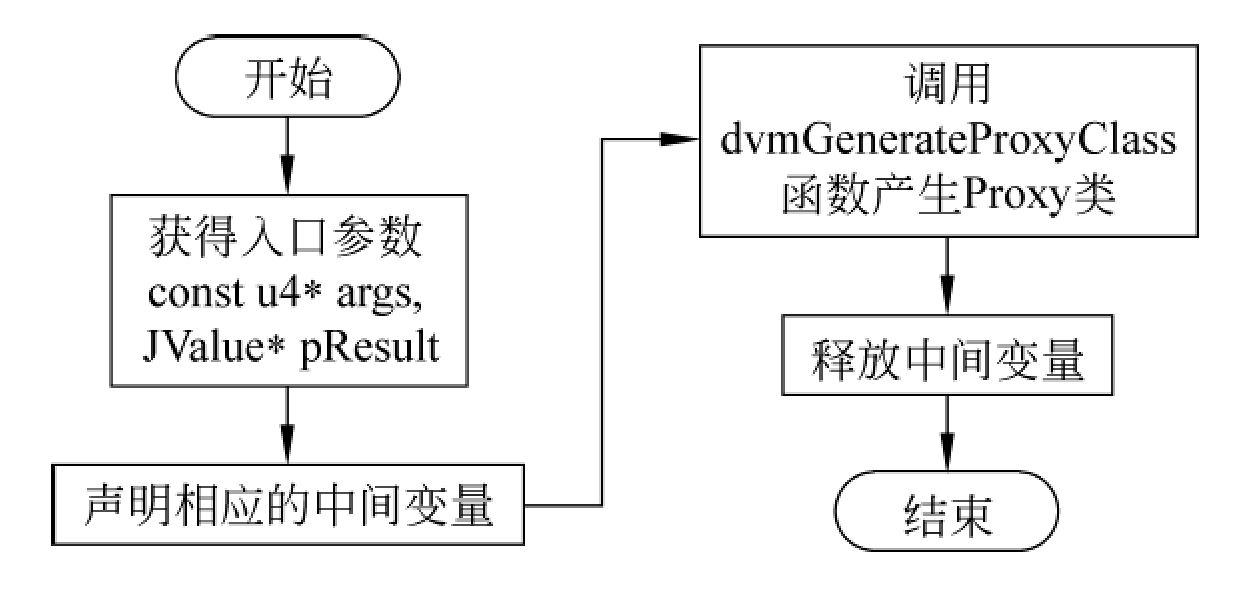


图 4.16　Dalvik_java_lang_reflect_Proxy_generateProxy 函数流程图

本函数是实现代理机制的重中之重，用于产生代理类，代码比较清晰明了，先观察一下代码。

代码清单 4.7　dalvik\vm\native\ java_lang_reflect_Proxy. cpp

```
static void Dalvik_java_lang_reflect_Proxy_generateProxy(const u4*  args,
```

```
        JValue* pResult)
{
    StringObject* str= (StringObject*) args[0];
    ArrayObject* interfaces= (ArrayObject*) args[1];
    Object* loader= (Object*) args[2];
    ClassObject* result;

    result= dvmGenerateProxyClass(str,interfaces,loader);
    RETURN_PTR(result);
}
```

主函数首先声明了变量用于接收入口参数，主函数通过调用 dvmGenerateProxyClass 函数完成产生一个 Proxy 类并将结果返回给 result。

至此，该函数的分析结束。

3. Dalvik_java_lang_reflect_Field_getField 详细分析

函数流程图如图 4.17 所示。

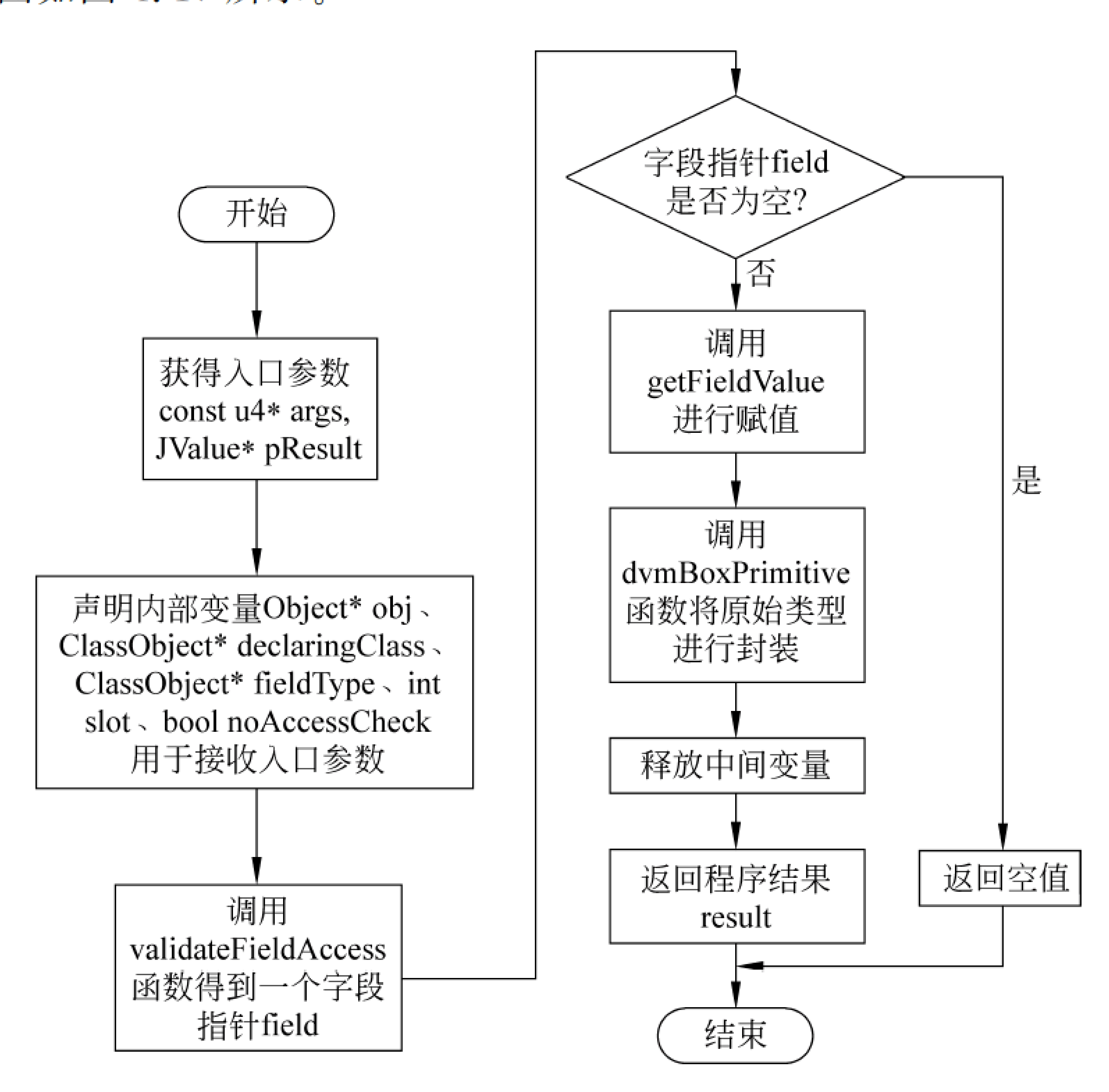


图 4.17 Dalvik_java_lang_reflect_Field_getField 函数流程图

本函数的主要功能是将一个原始字段进行封装。本函数的入口参数 args 为一个常量，pResult 为虚拟机参数。进入主函数体，首先函数将入口参数 args 转换成 Object 类的指针变量并赋值给 obj，再将 args 转换成 ClassObject 类的指针变量并赋值给 declaringClass，接下来按同样的方法将变量分别转换成相应的类型分别赋值给 fieldType、slot、noAccessCheck。之后函数体定义了一个字段指针 field，value 为虚拟机参数，定义了一个

数据对象指针为 result。

接下来函数调用 validateFieldAccess 函数，这个函数是用来验证访问字段并且返回一个指向结构体的字段 field。在获得函数 validateFieldAccess 返回的字段 field 之后，主函数对 field 进行非空判断。随后调用 getFieldValue 函数，此函数会将刚刚获得并判断是否为空的字段中的值赋给 value。最后函数调用 dvmBoxPrimitive 函数将字段进行封装，然后赋给 result。调用 dvmReleaseTrackedAlloc 函数，停止跟踪这个对象。返回刚刚获得的封装好的数据对象 result。

至此，完成了对函数 Dalvik_java_lang_reflect_Field_getField 的详细分析。

4. Dalvik_java_lang_reflect_Field_setField 详细分析

函数流程图如图 4.18 所示。

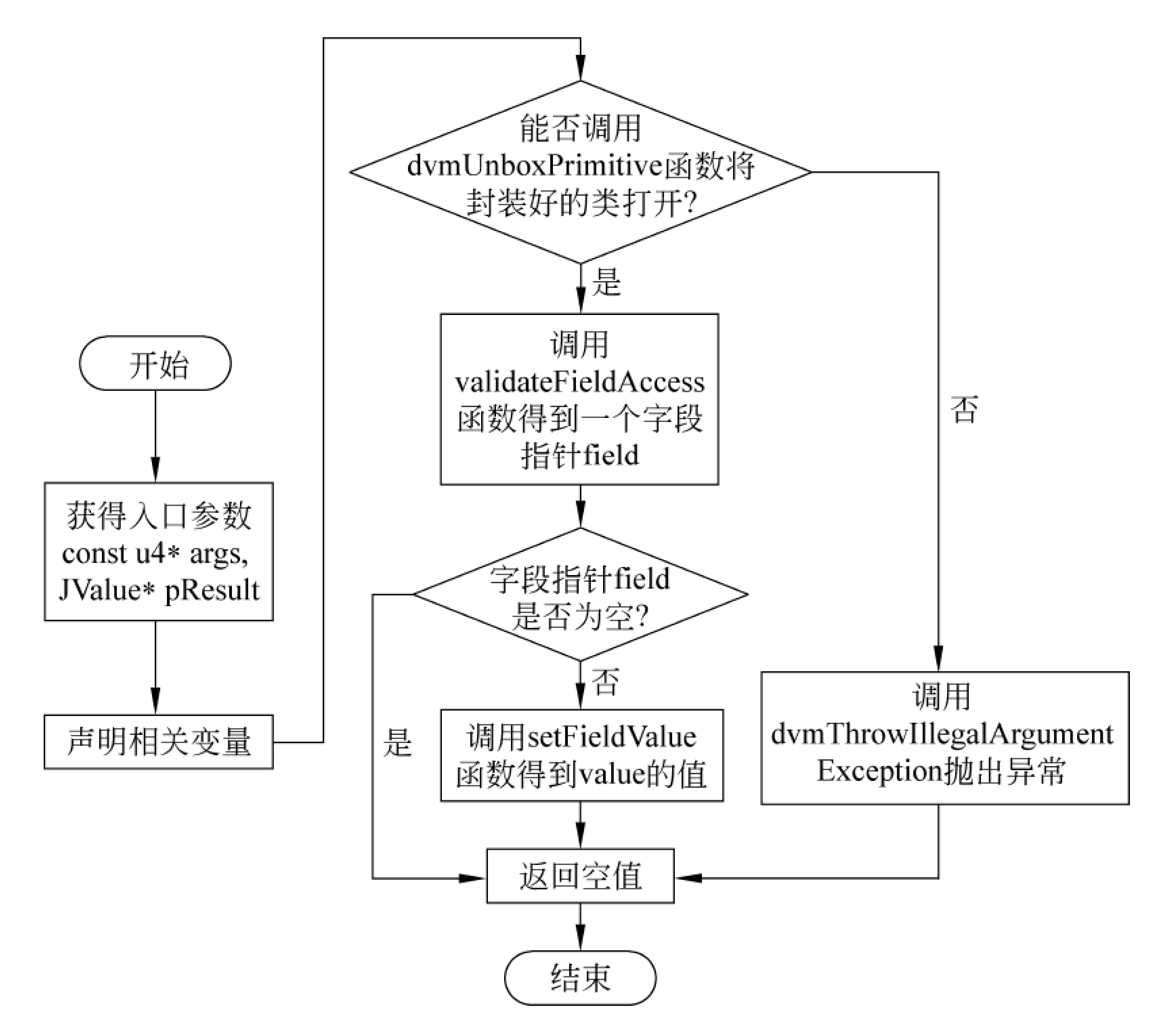


图 4.18　Dalvik_java_lang_reflect_Field_setField 函数流程图

本函数的主要功能是获得一个封装好的原始字段中的值。本函数的入口参数 args 为一个常量，pResult 为虚拟机参数。进入主函数体，首先函数将入口参数 args 转换成 Object 类的指针变量并赋值给 obj，再将 args 转换成 ClassObject 类的指针变量并赋值给 declaringClass，接下来按同样的方法将变量分别转换成相应的类型分别赋值给 fieldType、slot、noAccessCheck。之后函数体定义了一个字段指针 field，value 为虚拟机参数，定义了一个数据对象指针为 result。

将所有变量定义完成之后，主函数首先进行判断，调用 dvmUnboxPrimitive 函数将封装好的原始类型打开，如果正常打开则函数正常运行，如果打开失败则函数调用 dvmThrowIllegalArgumentException 抛出非法参数异常。

打开封装好的类之后再调用 validateFieldAccess 函数，这个函数是用来验证访问字段

并且返回一个指向结构体的字段 field。在获得函数 validateFieldAccess 返回的字段 field 之后主函数对 field 进行非空判断。如果返回的字段 field 不为空值，则调用 setFieldValue 函数将得到的字段的值返回。这样就成功地将一个封装好的字段中的值提取出来了。

至此，完成了对函数 Dalvik_java_lang_reflect_Field_setField 的详细分析。

5. dvmFindClassByName 详细分析

函数流程图如图 4.19 所示。

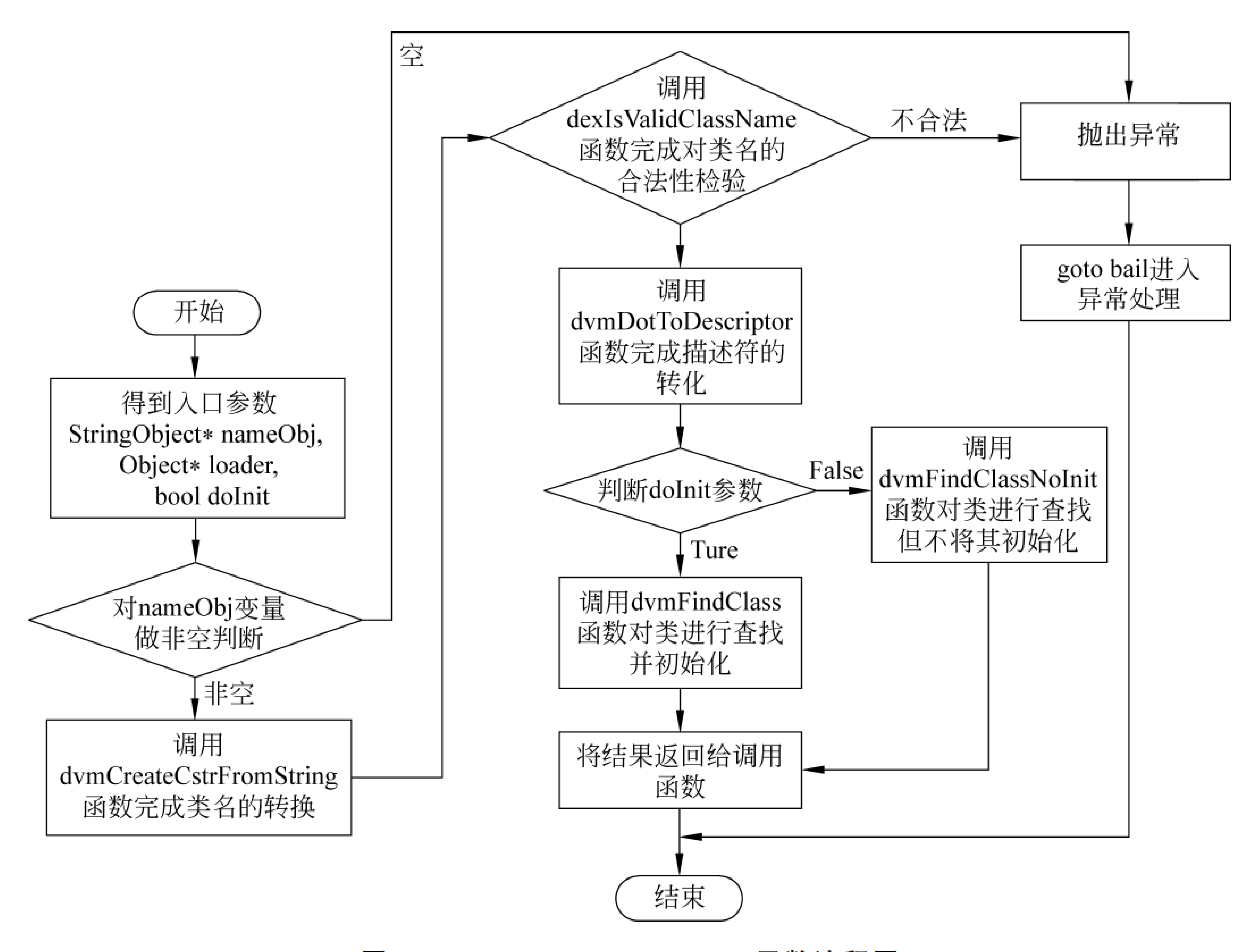


图 4.19　dvmFindClassByName 函数流程图

经过 JNI 的过渡作用，虚拟机内的 dvmFindClassByName 函数是静态 Java 方法 Class.forName 的底层实现函数，下面对这个函数进行详细的实现分析。

首先分析一下本函数入口参数 StringObject * nameObj——目标类类名（但在虚拟机运行时不是一个 string 类型的字符串），Object * loader——指定的类加载器，bool doInit——一个布尔类型的标示位，告诉虚拟机是否要对所找的类进行初始化。

进入函数体后，主函数 dvmFindClassByName 首先对目标类类名做非空判断，随后完成类名的字符型转化，主函数通过调用 dvmCreateCstrFromString 函数完成类名的转化并将这个字符型类名赋值给一个 char 型变量 name。在对类名进行转化后，主函数将会对这个类名进行验证，并且对它的格式进行转化（从 x. y. z 到 x/y/z），通过调用 dexIsValidClassName 完成类名正确性的验证并通过调用 dvmDotToDescriptor 函数完成格式的转化，并将转化后的类描述符赋值给 descriptor 变量。

主函数到这里就要开始它的主要工作了——根据一个正确的类描述符找到并加载这个

类。随后主函数会判断前面提到的布尔类型的标示位——提示是否要对加载好的类进行初始化,如果要求做初始化主函数会调用 dvmFindClass 函数(这个函数会根据描述符找到并加载,随后便对这个加载好的类进行初始化),否则就会调用 dvmFindClassNoInit(它的功能与 dvmFindClass 函数相近,只是不会对类进行初始化)。它们的返回值都是一个类对象实例 clazz。

到这里,主函数将会开始收尾工作,会对 clazz 变量做非空判断:如果为空,则证明查找加载失败,虚拟机会打出日志并抛出异常信息;如果非空,则表示运行正确,打印日志并返回 clazz 对象。

至此,完成了 dvmFindClassByName 函数的详细分析,具体的实现流程很清晰,其中涉及的一些具体的功能点函数就不在本报告中再做讨论。

6. dvmGetDeclaredConstructorOrMethod 详细分析

函数流程图如图 4.20 所示。

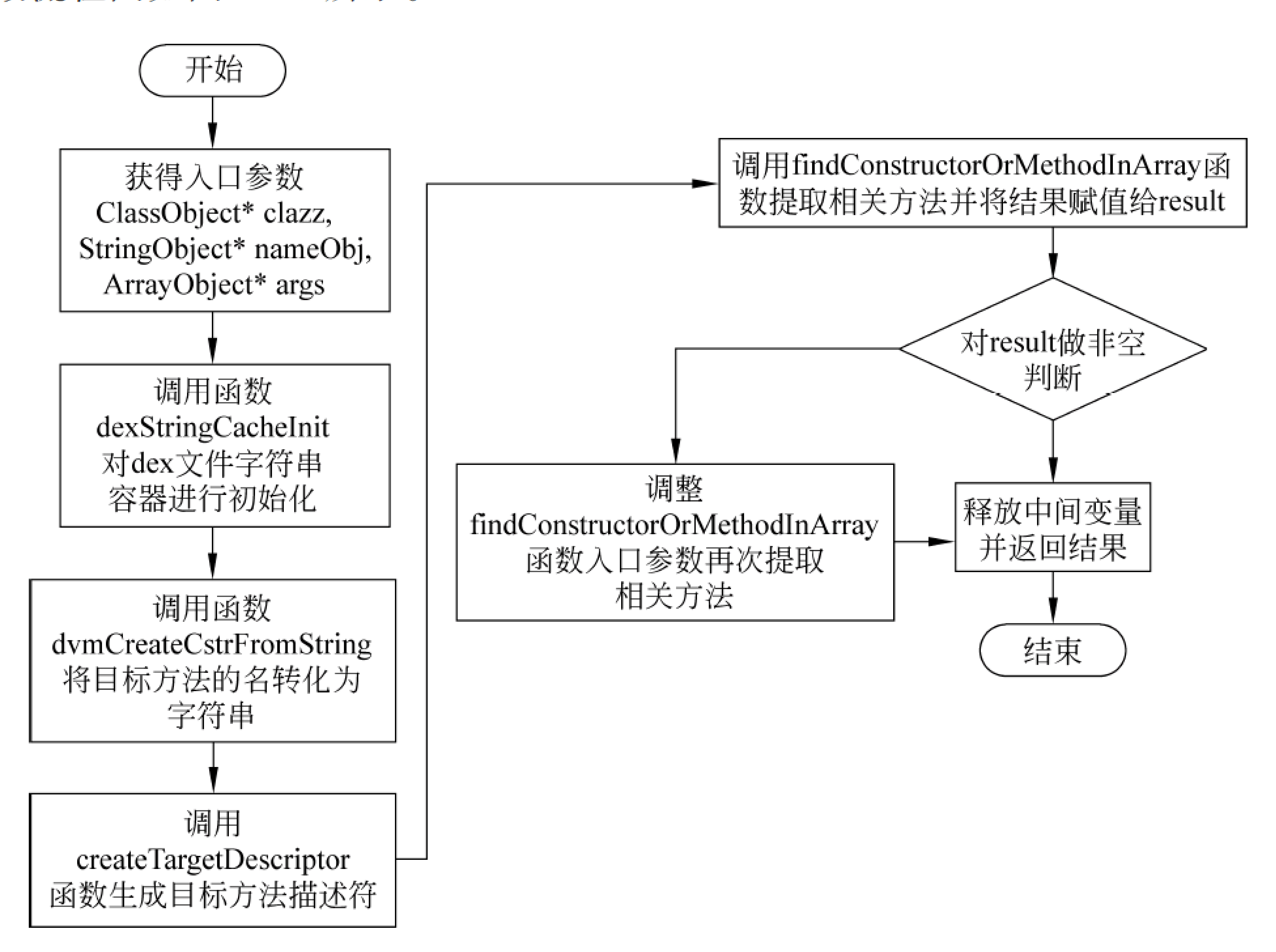


图 4.20 dvmGetDeclaredConstructorOrMethod 函数流程图

本函数是静态 Java 方法 Class.getDeclaredConstructorOrMethod 的底层实现函数,它的主要功能是根据一个指定的方法名将这个方法从一个类对象取出并返回给一个方法对象。

首先,看下它的入口参数,ClassObject * clazz——一个类对象实例,StringObject * nameObj——目标方法名,ArrayObject * args——相关参数。

进入函数体,主函数首先声明一个 Object 类型的指针,用于返回保存取得的方法,随后声明一个 DexStringCache 类型的结构体实例 targetDescriptorCache,用来封装 Dex 文件的一些字符串信息并且还会声明两个 char 型变量 name 以及 targetDescriptor,它们的作用在后文会点出。

进行相关变量的声明后，主函数调用 dexStringCacheInit 函数对前面声明的 targetDescriptorCache 进行初始化，随后便调用函数 dvmCreateCstrFromString 函数将目标方法的名转化为字符串并赋值给 name 变量，在取得 name 名后主函数调用 createTargetDescriptor 函数根据目标方法的参数生成目标方法的描述符，并将结果装入 DexStringCache 类型的结构体实例 targetDescriptorCache，随后便将 targetDescriptorCache 对象的一个成员变量 value 赋值给前面声明的 targetDescriptor。运行到这，主函数将要对方法进行提取，通过调用 findConstructorOrMethodInArray 函数再结合前面产生的各类目标方法的信息在类对象中查找并提取相关的方法，如果运行正常将方法返回给 result 变量。

至此，dvmGetDeclaredConstructorOrMethod 函数的详细分析结束。

7. dvmGetDeclaredMethods 函数详细分析

函数流程图如图 4.21 所示。

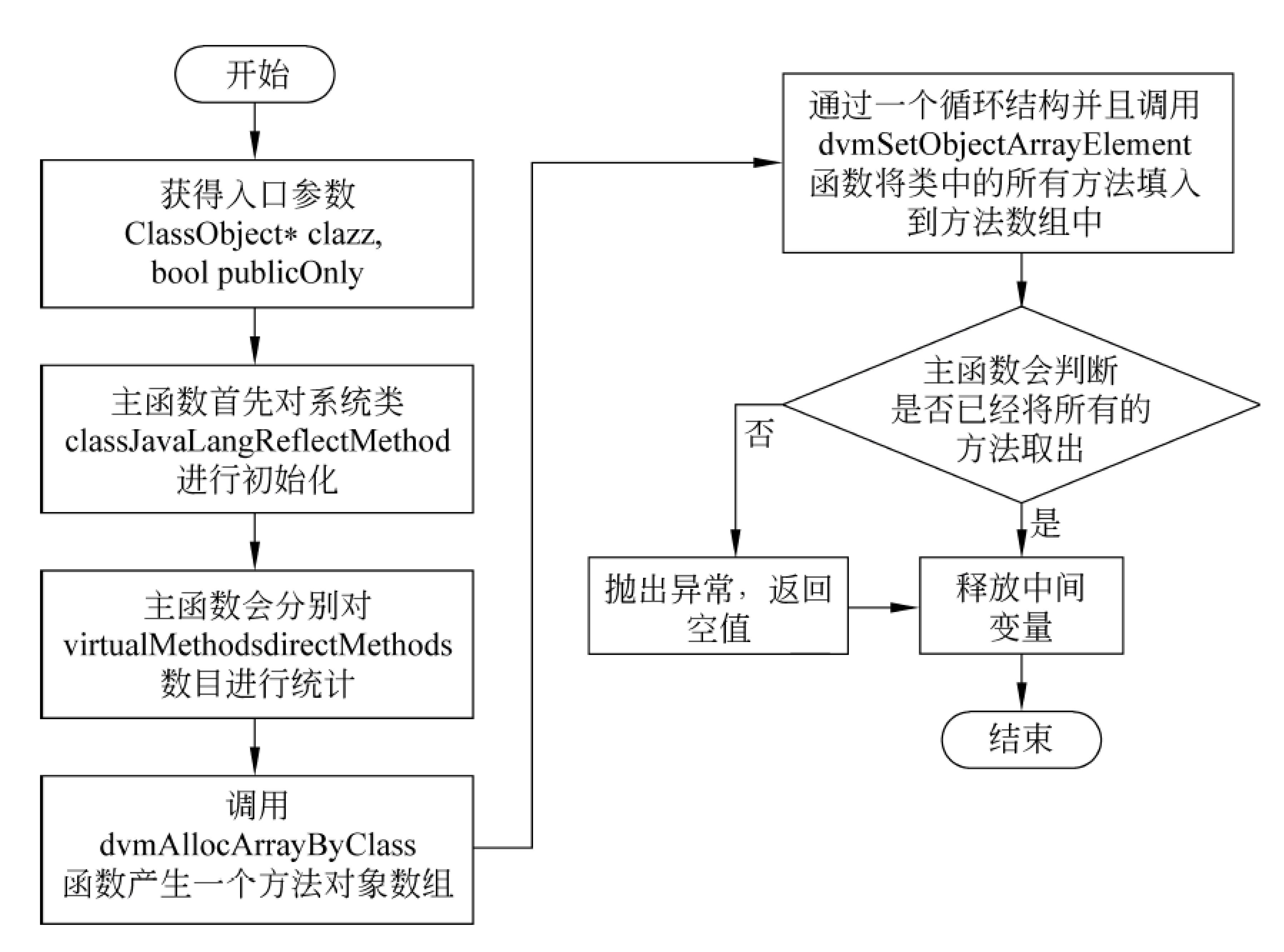


图 4.21 dvmGetDeclaredMethods 函数流程图

经过 JNI 的过渡作用，虚拟机内的 dvmGetDeclaredMethods 函数是静态 Java 方法 getDeclaredMethods 的底层实现函数，下面对这个函数进行详细的实现分析。

首先对函数的入口参数进行分析：ClassObject * clazz 表示一个已经加载到内存中的类对象，bool publicOnly 表示是否为 public 类型。进入函数体后，主函数首先对系统类 classJavaLangReflectMethod 进行初始化。

主函数随后对声明的方法进行统计，但虚拟机将 virtual Miranda methods 和 direct class/object constructors 等方法忽略。主函数首先将方法计数器 count 清零，随后获取类对象方法区指针，接下来顺次对类中的方法进行判断，主函数在这里利用一个循环结构完成对方法的遍历。

主函数到这里将会生成一个方法数组 methodArray，用来保存从类对象中取到的方法。随后主函数将要“填写”这个方法数组，也就是将声明的方法都保存到这个方法数组中，依然

忽略 virtual Miranda methods 和 direct class/object constructors 等方法，具体的读入流程如下：主函数首先调用 dvmCreateReflectMethodObject 函数生成方法对象实例 methObj，随后主函数调用 dvmSetObjectArrayElement 函数将当前方法读入本文在前面声明的方法数组 methodArray，最后主函数调用 dvmReleaseTrackedAlloc 释放掉用来保存方法的中间对象 methObj。这样就完成了将类对象中的一个方法保存到方法数组中，主函数在这里用到了循环结构，实现了对封装在类对象中所有方法的遍历。以上是对类对象中的虚方法进行录入保存的过程，对直接方法的处理方式完全一样，就不再赘述。

最后主函数将要进行收尾工作，它会检查是否已经将类对象中的全部方法全部遍历，如果正常则返回方法数组，否则释放已经获取的方法数组。

8．dvmGetDeclaredConstructors 详细分析

函数流程图如图 4.22 所示。

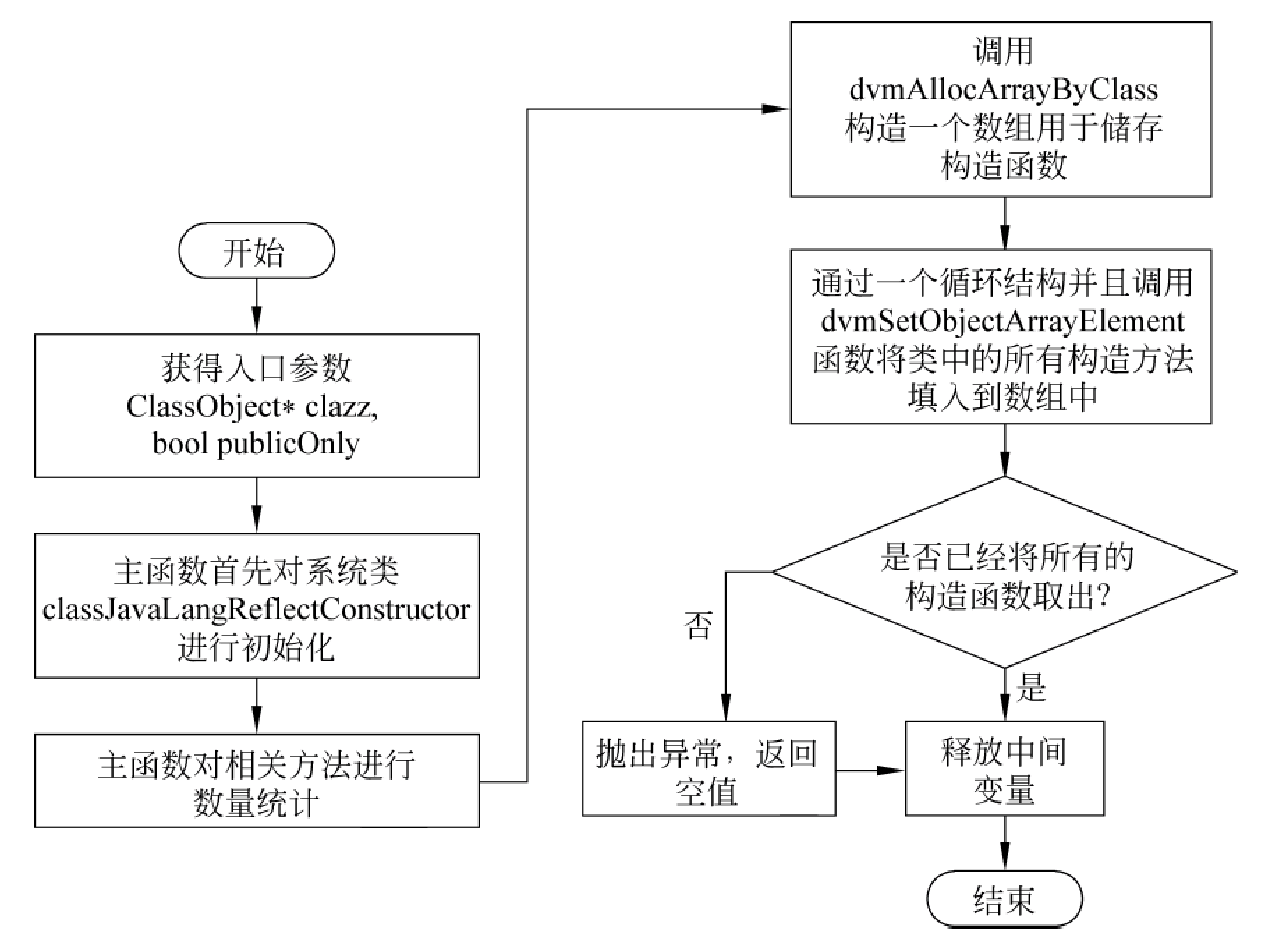


图 4.22　dvmGetDeclaredConstructors 函数流程图

经过 JNI 的过渡作用，虚拟机内的 dvmGetDeclaredConstructors 函数是静态 Java 方法 getConstructors、getDeclaredConstructors 的底层实现函数，下面对这个函数进行详细的实现分析。

首先对函数的入口参数进行分析：ClassObject * clazz 表示一个已经加载到内存中的类对象，bool publicOnly 表示是否为 public 类型。进入函数体后，主函数首先对系统类 classJavaLangReflectConstructor 进行初始化。

主函数首先将方法计数器 count 清零，随后获取类对象构造方法区指针，接下来顺次对类中的所有方法进行判断，主函数在这里利用一个循环结构完成对方法构造方法以及静态方法的遍历。

下一步主函数会生成一个数组 ctorArray，用来保存从类对象中取到的构造方法。在构

造函数数组生成好之后主函数遍历所有构造方法，将满足条件的字段填入上一阶段生成好的字段数组 ctorArray 中。函数的具体填入流程如下：首先调用 createConstructorObject 生成一个构造方法对象，然后判断是否为空，如果为空则调用 dvmReleaseTrackedAlloc 函数停止跟踪并返回空值；如果不为空则调用 dvmSetObjectArrayElement 函数将当前构造方法存入数组 ctorArray 中，最后程序调用 dvmReleaseTrackedAlloc 函数释放中间对象 ctorObj。通过循环结构实现了将所有满足条件的构造方法全部填入到数组 ctorArray 中。

最后主函数检查构造函数数组 ctorArray 长度是否等于构造函数对象的数量 ctorObjCount，如果相等就说明已经遍历了所有构造函数，此时程序返回获得的构造函数数组 ctorArray。

至此，完成了对 dvmGetDeclaredConstructors 函数的详细分析。

9. dvmGetDeclaredFields 函数详细分析

函数流程图如图 4.23 所示。

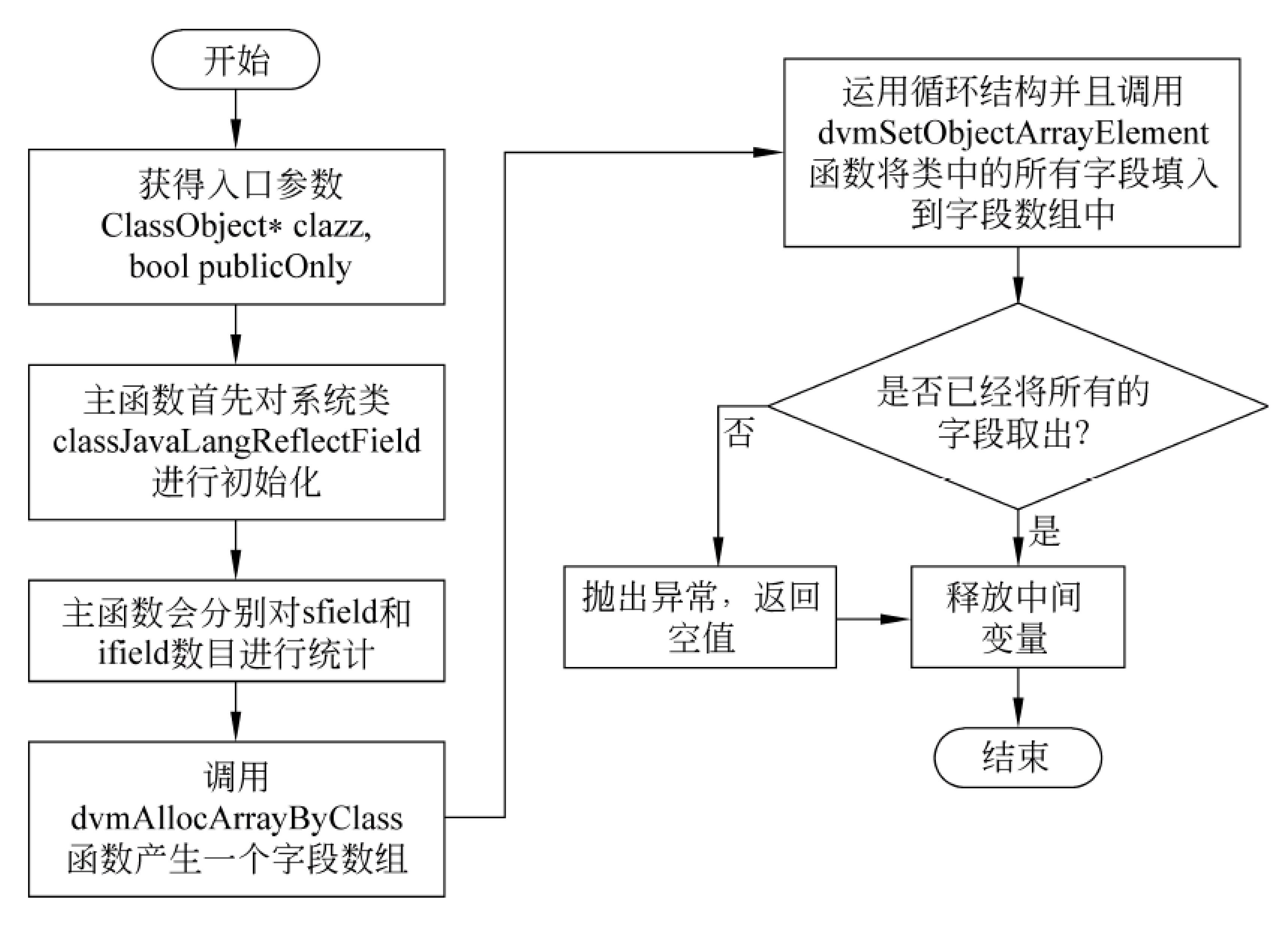


图 4.23 dvmGetDeclaredFields 函数流程图

经过 JNI 的过渡作用，虚拟机内的 dvmGetDeclaredFields 函数是静态 Java 方法 getDeclaredFields 和 getFields 的底层实现函数，下面对这个函数进行详细的实现分析。

首先对函数的入口参数进行分析：ClassObject * clazz 表示一个已经加载到内存中的类对象，bool publicOnly 表示是否为 public 类型。

进入函数体后，主函数首先对系统类 classJavaLangReflectField 进行初始化。

函数首先定义一个计数器来统计字段的数量。首先判断是否为 public 类型，然后利用两个循环体分别统计实例字段和静态字段的数量。实例字段和静态字段之和就是 count 值，它在下一阶段就作为生成的字段数组的长度。

主函数到这里将会生成一个字段数组 fieldArray，用来保存从类对象中取到的字段。在字段数组生成好之后主函数遍历所有字段，将满足条件的字段填入上一阶段生成好的字段数组 fieldArray 中。函数的具体的填入流程如下：函数首先调用 createFieldObject 生成

一个字段对象，然后判断是否为空字段，如果为空则释放所有数组；如果不为空则调用 dvmSetObjectArrayElement 函数将当前字段存入字段数组 fieldArray 中，最后程序调用 dvmReleaseTrackedAlloc 函数释放中间对象 field。通过循环结构实现了将所有满足条件的静态字段全部填入到字段数组 fieldArray 中。

在存储实例字段时所运用的方法与静态字段完全相同，就不再赘述了。

最后主函数检查字段数组长度是否等于字段数，如果相等就说明已经遍历了所有数组，此时程序返回获得的字段数组，否则释放已经获取的字段数组。

至此，完成了对 dvmGetDeclaredFields 函数的详细分析。

4.6　模块内部函数调用关系

反射机制模块内部函数调用关系主要集中在两个方面：①反射机制本地方法接口对反射机制实际执行函数的调用；②反射机制实际执行函数内部对各个功能点函数的调用。由于反射机制主要面向对 Java 层 API 的实现，因此根据"三层结构"的实现框架，单个接口的调用关系相对简单，但由于核心库中反射 API 较多并且各个接口的调用关系十分相似，因此，就不在此一一展示，下面主要根据两方面的调用关系，举出具有代表性的实例以展示反射机制内部的调用机制的具体特征。

4.6.1　反射机制本地方法接口对反射机制实际执行函数的调用

在这里主要通过一个本地方法的调用关系示例以展示本地方法对反射机制执行函数的调用关系。

Class 类中 getConstructor 方法（用于取得指定的类构造函数）的本地方法接口 Dalvik_java_lang_Class_getDeclaredConstructorOrMethod 对下层各个执行函数的调用关系如图 4.24 所示。

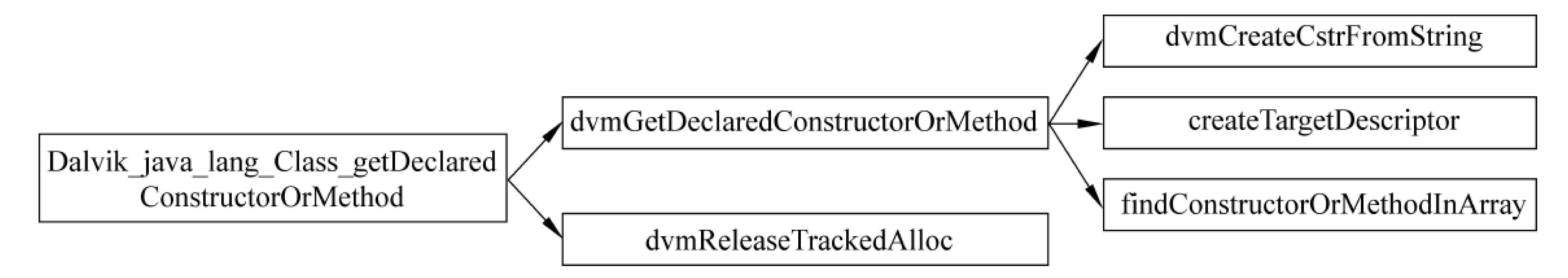


图 4.24　Dalvik_java_lang_Class_getDeclaredConstructorOrMethod 函数调用

从图 4.24 中可以看到，本地方法通过调用 dvmGetDeclaredConstructorOrMethod 函数完成取得指定构造函数或方法的目标。

通过以上例子较直观地展示了反射机制本地方法接口对反射机制实际执行函数的调用关系。

4.6.2　反射机制实际执行函数内部对各个功能点函数的调用

在这里主要通过三个本地方法的调用关系示例以展示本地方法对反射机制执行函数的调用关系。

（1）createFieldObject 函数（用于创建一个字段对象）对下层各个执行函数的调用关系如图 4.25 所示。

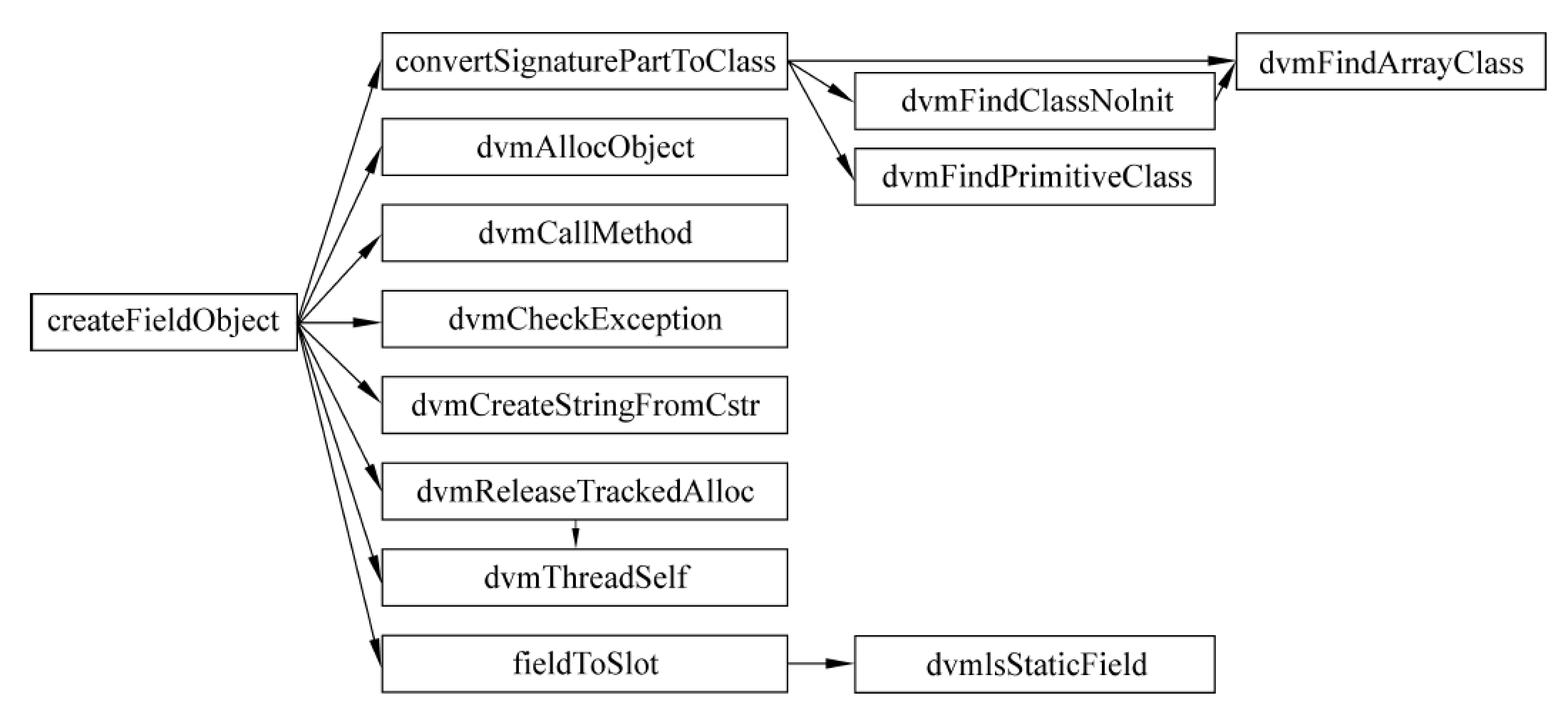


图 4.25　createFieldObject 函数调用

从图 4.25 中可以看到，createFieldObject 函数通过调用一系列功能点函数实现了其目标功能，其中 dvmAllocObject 函数用于实现关键步骤——创建一个字段对象。

（2）createConstructorObject 函数（用于创建一个构造函数实例对象）对下层各个执行函数的调用关系如图 4.26 所示。

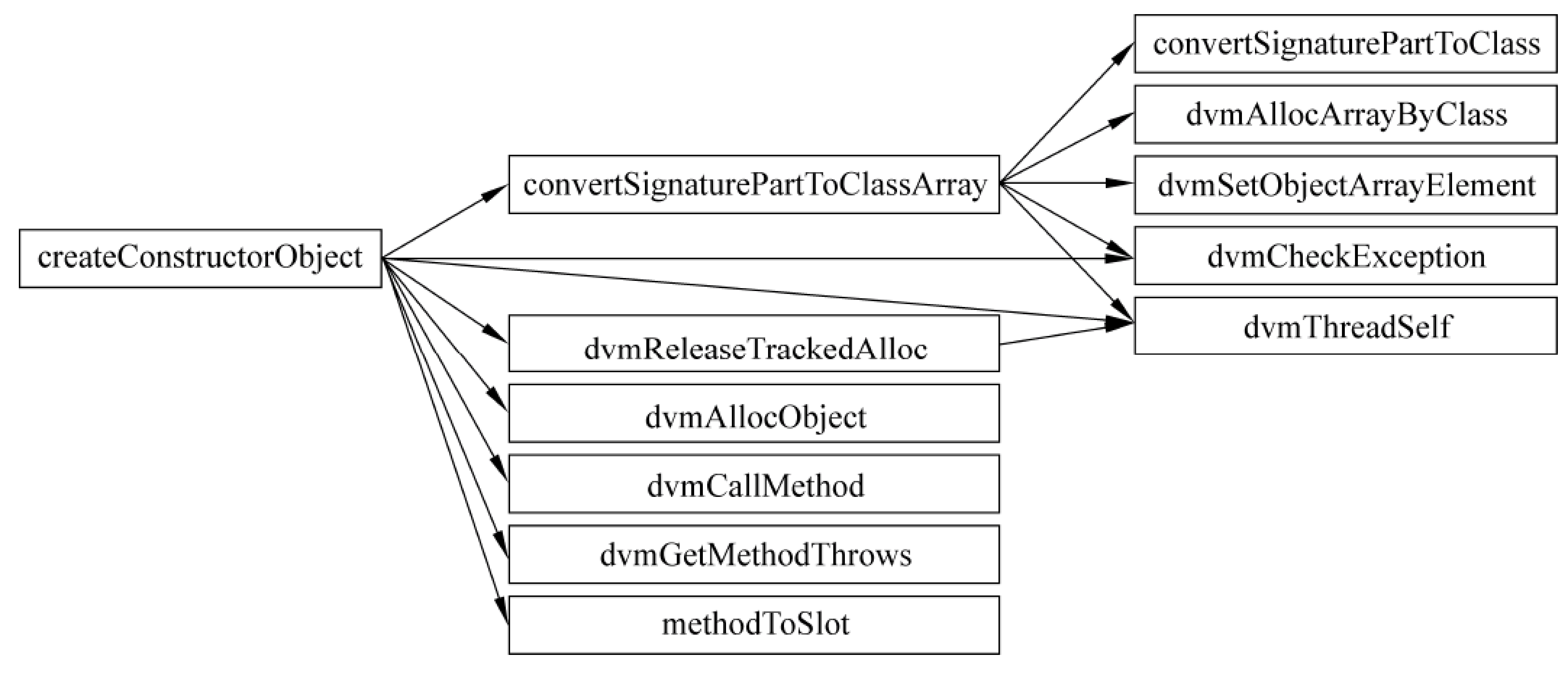


图 4.26　createConstructorObject 函数调用

从图 4.26 中可以看到，createConstructorObject 函数通过调用一系列功能点函数实现了其目标功能，其中 dvmAllocObject 函数用于实现关键步骤——创建一个构造函数对象。

（3）dvmCreateReflectMethodObject 函数（用于创建一个方法类型实例对象）对下层各个执行函数的调用关系如图 4.27 所示。

从图 4.27 中可以看到，dvmCreateReflectMethodObject 函数通过调用一系列功能点函数实现了其目标功能，其中，dvmAllocObject 函数用于实现关键步骤——创建一个方法实例对象。

通过以上三个例子较直观地展示了反射机制实际执行函数内部对各个功能点函数的调

用关系。

点拨　以上便是对模块内部调用关系的实例分析，总的来说，反射机制相对于虚拟机中的其他功能模块要更接近上层应用，实际上就是对 Java 类库中反射 API 的底层实现，然而这种实现是以"三个层次"结构为基础的。

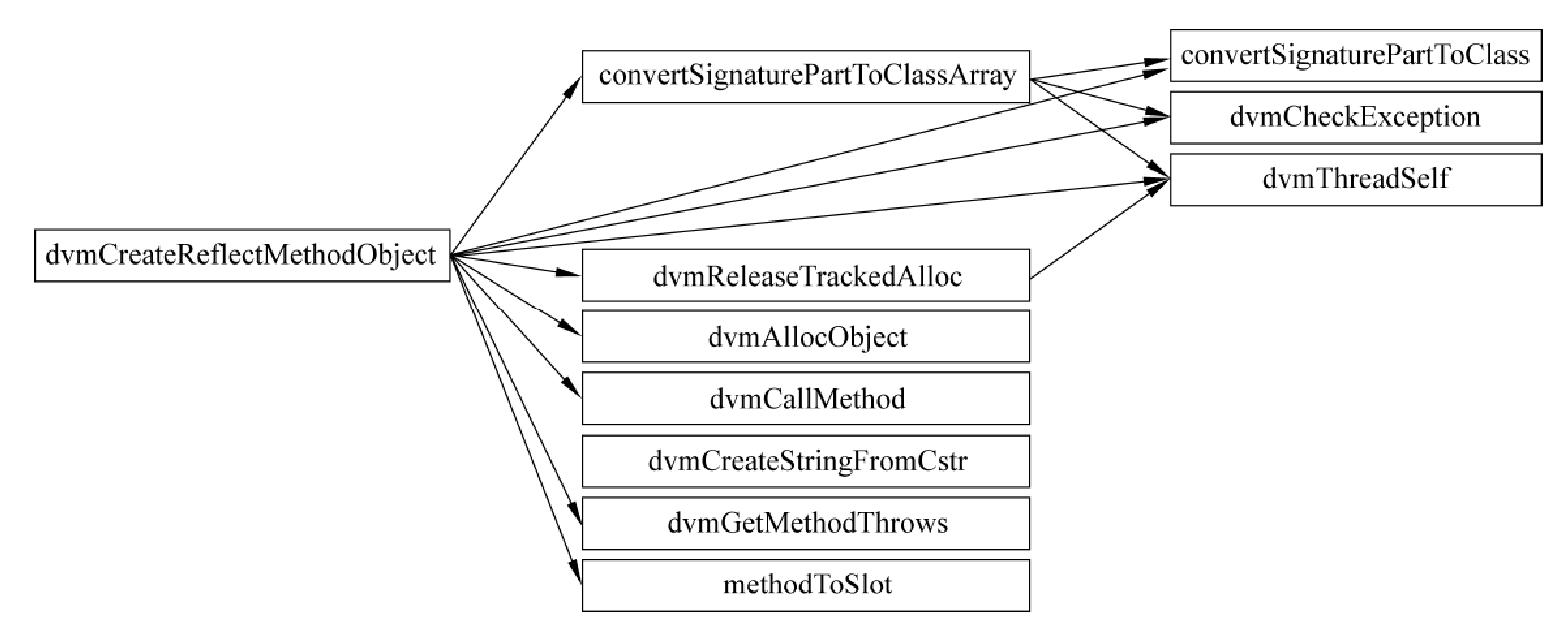


图 4.27　dvmCreateReflectMethodObject 函数调用

小　　结

本章对反射机制原理进行了详细分析，首先分析 API 结构，总结出反射机制 API-本地方法-底层函数的接口"三层结构"体系，具体说明反射机制在 Dalvik 虚拟机中的实现流程。随后对反射机制中所涉及的核心类进行详细分析，说明类中具体方法的实现流程，并绘制流程图。最后对核心函数进行详细分析，逐行解读函数作用、功能、原理和流程，并绘制函数流程图。

第5章 解释器模块的原理及实现

本章主要内容

✍ Dalvik 是如何执行字节码的？

✍ 如何克服不同硬件平台带来的程序移植性问题？

✍ 尽可能地复用代码和模块化设计的目的是什么？

解释器是 Dalvik 虚拟机的执行引擎，它负责解释执行 Dalvik 字节码。在字节码加载完毕后，Dalvik 虚拟机调用解释器开始取指解释字节码，解释器跳转到解释程序处执行。在 Android 4.04 版本中，解释器共有两种，称作移动型(Portable)解释器和快速型(Fast)解释器，分别采用 C 语言实现和汇编语言实现。

5.1 概述

快速型(Fast)解释器是由 Google 公司为提高运行效率用汇编语言重写的解释器。系统使用 Portable 解释器还是 Fast 解释器是由 Android 开机时就已经决定的。在 Fast 解释器中，汇编的实现是针对特定平台的，而且在执行完成之后可自动取指并跳到相应的地址开始解释，所以有比较高的效率，也是作为默认的解释器。同时，为了克服可移植性低的缺点，也提供了 Fast 解释器的 C 语言实现。

Portable 解释器是最初采用的方案，虽然效率较 Fast 解释器低，但是有移植性好的特点。在 Android 4.04 版本中采用 GCC 的 Threaded 技术优化后，Portable 解释器的效率有了质的提升。

在设计实现上，Dalvik 虚拟机解释器采用的是模块化的设计和实现，有高内聚低耦合的特点。所以可移植性非常好，对于不同的平台只需要修改少量的代码和配置文件，即可快速得到相应平台的代码。同时，在 Fast 解释器汇编语言中，也有部分代码调用 C 语言实现，减轻实现的难度；Fast 解释器 C 实现中，可以直接调用 C 代码，以提高解释器的效率。

5.2 解释器执行原理

获取字节码并分析与解释执行是 Dalvik 虚拟机解释器的主要工作。Dalvik 虚拟机的入口函数是 vm/interp 下的 dvmInterpret 函数。外部通过调用 dvmInterpret 函数进入解释器执行，Android 中解释器与线程是一一对应的，外部调用解释器解释指令的流程图如图 5.1 所示。



图 5.1　外部调用

在外部函数调用解释器后，解释器执行的主要流程有如下几个步骤。

(1) 初始化解释器执行环境；

(2) 根据系统参数得到执行 Fast 解释器还是 Portable 解释器；

(3) 跳转到相应的解释器执行；

(4) 取指及指令检查；

(5) 执行字节码对应的程序段。

dvmInterpret 函数作为解释器的入口函数，主要完成了整个流程的前三部分，执行流程如图 5.2 所示。

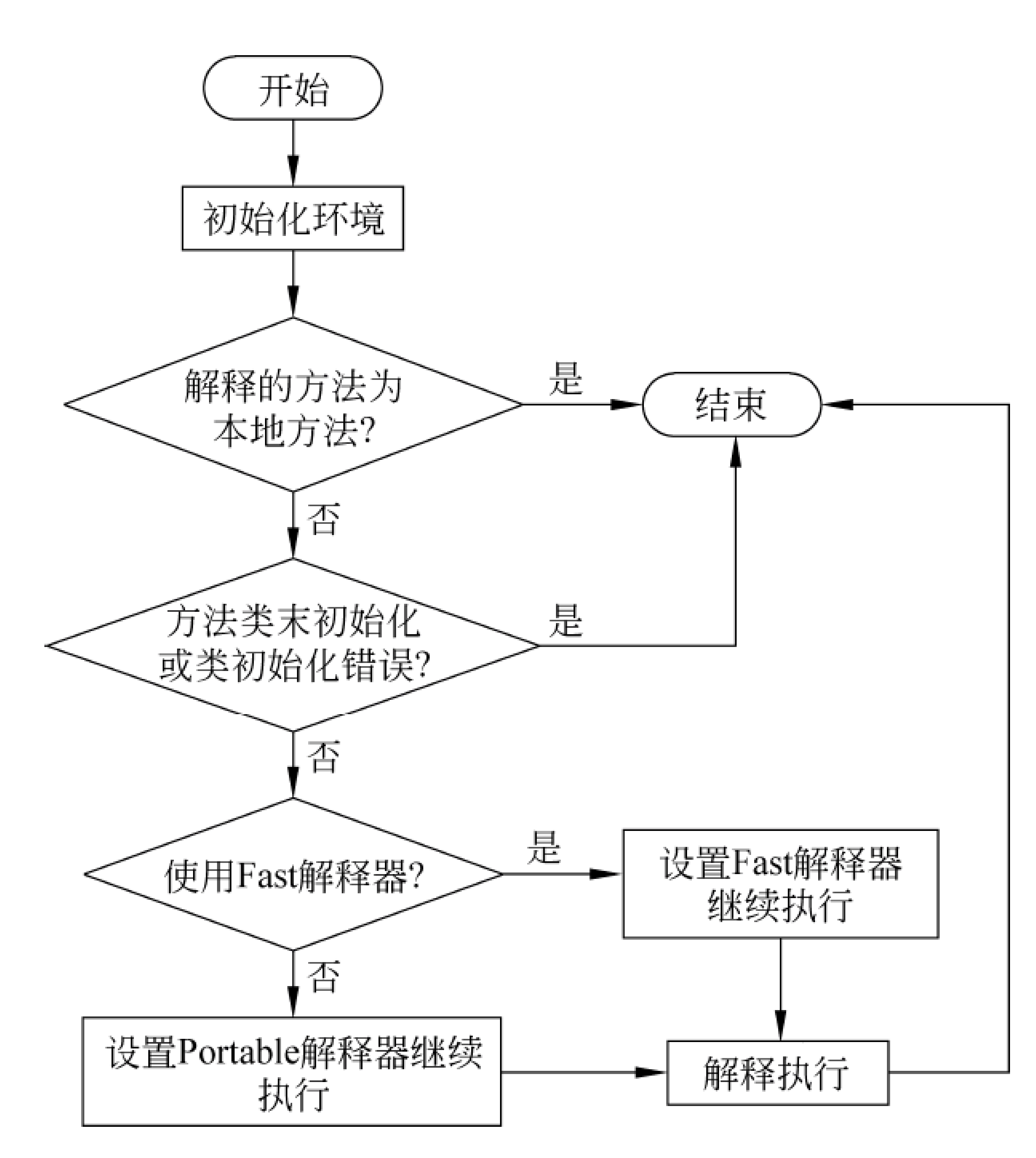


图 5.2　解释器入口函数流程图

(1) 解释器执行环境的初始化。主要是对解释器的变量进行初始化。解释器得到将要解释的方法的指针 method，以及保存当前栈指针 self→interpSave.curFrame 和程序计数器 method→insns。

(2) 判断将要解释的方法是否合法。主要进行两个方面的判断，分别判断方法是否是本地方法 dvmIsNativeMethod(method)和判断方法的类是否初始化 CLASS_INITIALIZING 以及初始化是否失败 CLASS_ERROR。

(3) JIT 环境的设置(如果有 JIT)。

(4) 选择相应的解释器(比如 Fast 解释器或 Portable 解释器)来执行。根据系统参数的不同，选择不同的解释器来执行解释。需要判断执行模式 gDvm.executionMode，如代码清单 5.1 所示。

代码清单 5.1 dalvik/vm/interp/Interp.cpp:dvmInterpret()

```
if (gDvm.executionMode==kExecutionModeInterpFast)
        stdInterp=dvmMterpStd;
#if defined(WITH_JIT)
    else if (gDvm.executionMode==kExecutionModeJit)
        stdInterp=dvmMterpStd;
#endif
    else
        stdInterp=dvmInterpretPortable;
```

通过检查 gDvm.executionMode 的值来选择执行模式。这个值是在 Dalvik 虚拟机启动时，通过解析命令行参数获得的。值有三种情况：①kExecutionModeInterpFast，代表使用 Fast 解释器来执行字节码解析，此时会调用 dvmMterpStd 函数进入 Fast 解释器执行；②kExecutionModeJit，代表使用 JIT(即时编译)来编译执行，使用的解释器是 Fast 解释器；③当系统判断执行模式非以上两种时，则会调用 dvmInterpretPortable 函数来使用 Portable 解释器执行字节码解析。

虽然执行模式有三种，但实际上解释器的解释模式只有两种，分别是 Fast 和 Portable，JIT 机制只能配合 Fast 解释模式使用。所谓 Fast 解释器和 Portable 解释器，顾名思义，Fast 解释器的主要特点是快速，而 Portable 解释器的拿手好戏则是可移植性。虽然，宏观上，这两种解释器的解释方式是类似的，但在细节上，其执行流程和实现原理有一些不同。下面就两种解释器分别做分析。

5.3 Portable 解释器实现分析

Portable，英文解释是指“便携式的，轻便的”，顾名思义，在 Dalvik 中，Portable 也是便携式的，其实现不依赖底层硬件平台，比如针对 ARM 和 x86，都可以使用。究其原因，主要是因为其使用纯 C 实现的。其主体是一个巨大的 C 函数。在任何支持 gcc 的系统里都可以编译，其具体实现依赖于 GCC 的 Threaded Code 技术。

5.3.1 字节码解析原理

根据解释器的功能，可以想到的最简单的模型就是用一个大的 switch 语句，对每条指令进行判断，然后 case 到相应的代码进行解释，解释完成后又要回到 switch 顶部，如代码清单 5.2 所示。

代码清单 5.2 解释器模型

```
while (insn) {
    switch (insn) {
        case NOP:
            break;
        case MOV:
            do something; break;
        ...
```

```
            case OP:
                do something; break;
            default:
                do something;
        }
        取指;
    }
```

然而当解释完成一条指令后，再重新判断指令类型是个昂贵的开销。因为对于每条指令，都将从 switch 顶部开始判断，也就是从 NOP 指令开始判断，直到找到相应的指令为止，这使得解释器的执行效率十分低下。

Dalvik 对此的解决方案是采用了 GCC 的 Threaded Code 技术，可以获得极高的分发效率。每条指令都有一个对应的标签(label)，标签标示的是该指令解释程序的开始。在每条指令的解释程序末尾，有取指动作，可以取得下一条要执行的指令，根据指令的不同，使用 threaded 机制，可以 goto 到相应的标签处执行解释程序，避免了重新跳转到 switch 顶部重新 case 这个昂贵的操作。

那 GCC 的 Threaded Code 技术又有什么特点呢？在 C 文件中，可以使用“&&”获得函数或函数内部标签地址，而 goto 语句可以直接跳转到这些地址。通常情况下是设置一个静态数组，用来存储标签地址。例如：

```
static void * array[]= {&&foo,&&bar,&&hack};
```

从而可以根据索引选择一个标签并跳转，如：

```
goto * array[i];
```

可以看到标签的使用和 switch 有点类似，但是 switch 语句能更清楚地表达含义，所以不到万不得已的时候，尽量不要使用这个技术。

读者应该已经明白，解释器是如何从 GCC 的这项技术中获益的了。不错，Portable 解释器的实现使用了标签数组，由此达到了极高的分发(dispatch)效率。

点拨 Threaded Code 技术也叫 Label as Value，具体的信息可以查看 GCC 的官方帮助文档：http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Labels-as-Values.html#Labels-as-Values。

Dalvik Portable 维护了一个静态数组，用来存储各个字节码解释程序对应的标签地址。其具体以一个宏来定义，如代码清单 5.3 所示。

代码清单 5.3 dalvik/libdex/DexOpcodes.h:DEFINE_GOTO_TABLE

```
#define DEFINE_GOTO_TABLE(_name)                          \
    static const void* _name[kNumPackedOpcodes]= {        \
        H(OP_NOP),                                        \
        H(OP_MOVE),                                       \
        H(OP_MOVE_FROM16),                                \
        ...                                               \
    };
```

DEFINE_GOTO_TABLE 宏在使用时，将定义一个静态数组。在其中，根据 C 语言中

宏的定义，_name 将被具体名称代替；而内部元素也由宏控制。在 Portable 版解释器实现中有如代码清单 5.4 所示的宏定义。

代码清单 5.4　vm/mterp/out/InterpC-portable.cpp：H(_op)

```
#define    H(_op)    &&op_##_op
```

该宏中的"##"在 C 语言里被称为连接符，用来连接两个 token。在预处理的时候，Goto Table 中的元素将会被做相应的替换。以 H(OP_NOP)为例，预处理后将会被替换为 &&op_OP_NOP。根据前面叙述过的 Threaded Code 技术，&&op_OP_NOP 是取标签 op_OP_NOP 的地址。由此可以看到，静态数组中存储的将是各个 Label 的地址。

每一个 Label 都对应于一段相应的指令解释程序。标签处的地址即是解释程序的开始地址。由此，可以为 Goto Table 中的元素和前述的指令解释程序的实现建立一一对应的关系，Goto Table 中每一项元素对应了相应解释程序的开始地址。

在解释器开始执行时，通过 DEFINE_GOTO_TABLE(handlerTable)宏的执行建立了如图 5.3 所示的一张名为 HandlerTable 的表。我们将这一过程称为 GOTO Label 的绑定。在图中，可以看到 HandlerTable 中每一个元素都指向了一个 Label。每个 Label 都对应了一段解释程序。

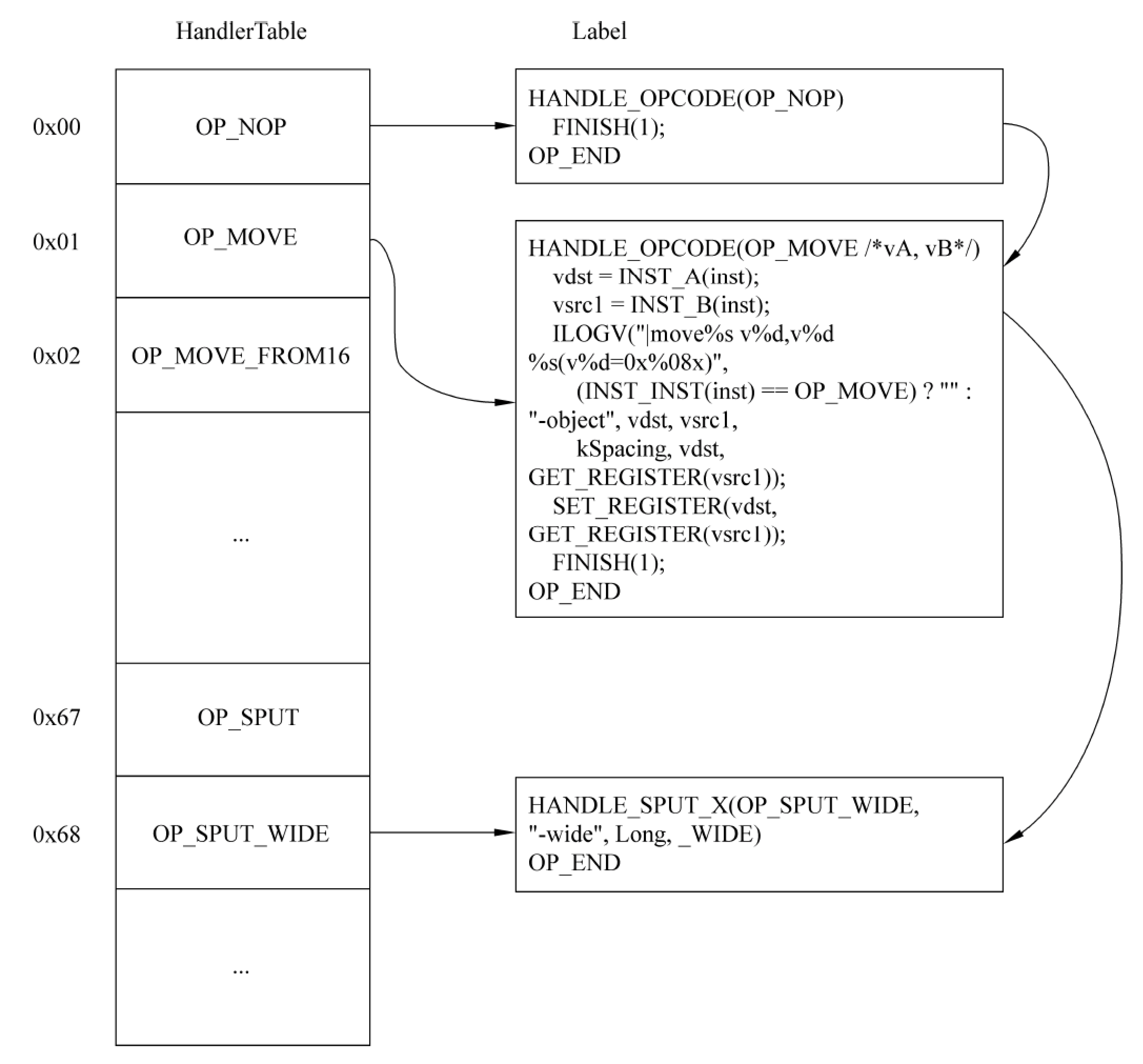


图 5.3　Portable 解释器指令解释程序构造图

同时，如何根据指令得到对应的 Label 地址呢？在 Dalvik 中，利用 Opcode-gen 工具生成了每个字节码在 HandlerTable 中的索引号，如代码清单 5.5 所示。如图 5.3 中左侧的十六进制数字。比如对于 NOP 操作，对应的索引号为 0x00；MOVE 操作对应的索引号是 0x01。Dalvik 在解析字节码时，根据操作码的类型，可以得到指令的索引号，由索引号索引 HandlerTable，因而实现了 Switch 中的 case 操作。

代码清单 5.5　dalvik/libdex/DexOpcodes.h：Opcode

```
enum Opcode {
    OP_NOP                                  = 0x00,
    OP_MOVE                                 = 0x01,
    OP_MOVE_FROM16                          = 0x02,
    OP_MOVE_16                              = 0x03,
    OP_MOVE_WIDE                            = 0x04,
    …
    OP_SPUT_OBJECT_VOLATILE_JUMBO           = 0x1fe,
    OP_THROW_VERIFICATION_ERROR_JUMBO       = 0x1ff,
};
```

5.3.2　字节码指令解释流程

在明白实现解释器的基本模型后，下面来看下 Dalvik Portable 的执行流程。Portable 解释器的具体执行流程如图 5.4 所示。

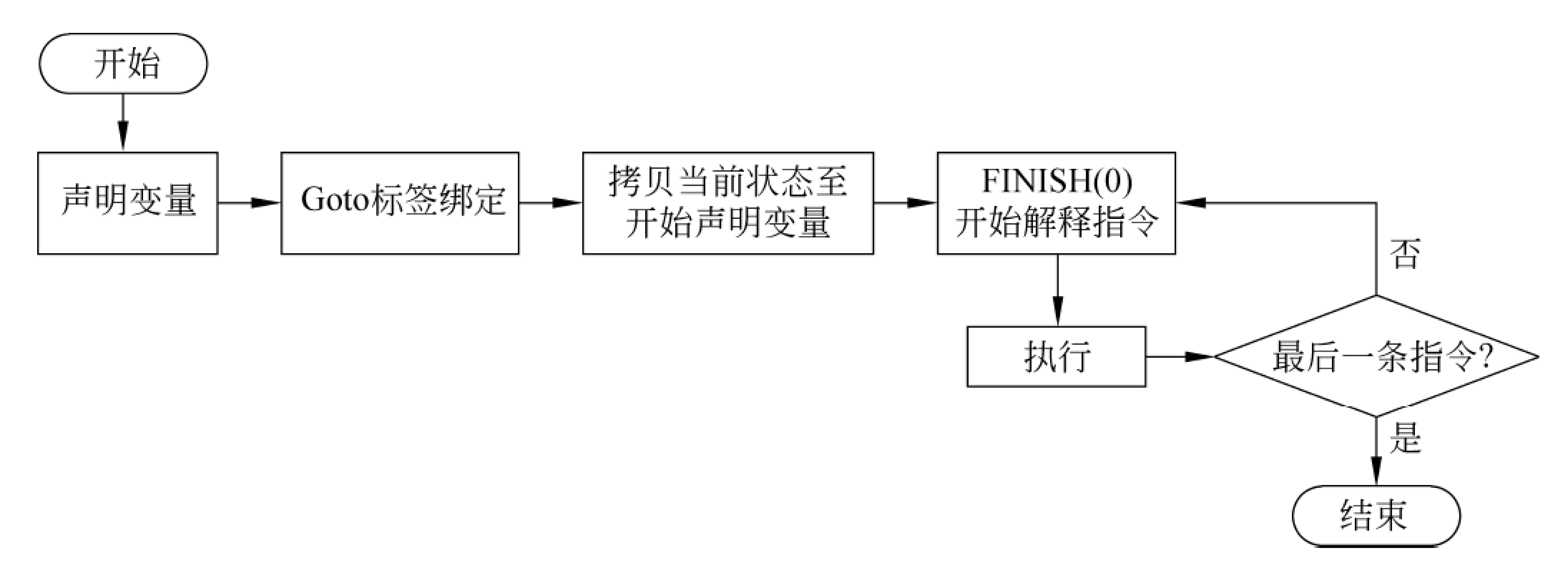


图 5.4　Portable 解释器解析流程图

如流程图中所示，在解释器开始解释前，需要设置一些环境，包括声明变量、Goto Label 绑定以及拷贝当前的状态。

首先进行相关变量的声明：保存当前正在解释的方法 curMethod、程序计数器 pc、栈帧指针 fp、当前指令 inst、指令译码的相关部分包括保存寄存器值 vsrc1，vsrc2，vdst、设置方法调用 methodToCall 等。

通过 DEFINE_GOTO_TABLE(handlerTable)这一宏定义进行 GOTO Label 的绑定，获取并拷贝 self→interpSave 里保存的当前状态，包括方法 method、程序计数器 pc、堆栈帧 curFrame、返回值 retval、要分析的 Dex 文件的类对象信息 curMethod→clazz→pDvmDex 等到已声明的变量，如代码清单 5.6 所示。

代码清单 5.6 dalvik/vm/mterp/out/InterpC-portable.cpp:dvmInterpretPortable()

```
curMethod=self->interpSave.method;
pc=self->interpSave.pc;
fp=self->interpSave.curFrame;
retval=self->interpSave.retval;
methodClassDex=curMethod->clazz->pDvmDex;
```

最后通过调用 FINISH(0)来取得第一条指令开始执行字节码解析。其中，除去状态的备份以及一些例行检查判断外，最重要的两项是 Goto 标签绑定和 FINISH 宏的执行。调用 FINISH 宏开始取指跳转到解释程序执行，在解释程序的最后进行取指，跳转执行，直到执行到最后一条指令。

在设置好环境后，开始进行正常的解释程序。正如图 5.3 中所示，在 Portable 解释器中，每一条指令对应的解释程序都对应了一个标签，每一个标签指明了对应的解释程序段。在 Dalvik Portable 中，解释程序是由一系列宏定义控制，以对应的 Label 来表示，以 NOP 操作为例，该操作的定义如代码清单 5.7 所示。

代码清单 5.7 dalvik/vm/mterp/out/InterpC-portable.cpp: HANDLE_OPCODE(OP_NOP)

```
HANDLE_OPCODE(OP_NOP)
    FINISH(1);
OP_END
```

HANDLE_OPCODE(OP_NOP)表示对应的是 OP_NOP 操作，紧接其后的是解释程序的具体实现。到 OP_END 结束。而在 Portable 中，所有的解释程序都由 C 语言编写。NOP 操作中的 HANDLE_OPCODE、FINISH 和 OP_END 都是宏定义。

其中，HANDLE_OPCODE 宏定义如代码清单 5.8 所示。

代码清单 5.8 dalvik/vm/mterp/out/InterpC-portable.cpp:HANDLE_OPCODE(_op)

```
#define    HANDLE_OPCODE(_op)    op_##_op:
```

注意其中的“:”，经过宏替换后，指明了这是一个 Label。因此，对于 NOP 指令，HANDLE_OPCODE(OP_NOP)将被替换为

```
op_OP_NOP:
```

每一个 HANDLE_OPCODE 都需要和 OP_END 成对出现。在此处，OP_END 的宏定义如下所示：

```
#define OP_END
```

注意其定义，没有错，OP_END 的具体定义什么都没有！也就是说，经过预处理后，OP_END 将消失。

因此，到目前为止，经过预处理后，NOP 指令的解释程序将如下所示。以 op_OP_NOP 标签指明解释程序地址。

```
op_OP_NOP:
```

```
    FINISH(1);
```

对于 NOP 指令,其完成的工作就是什么都不做。因此,对应的解释程序就是直接取下一条将要解释的指令,也就是 FINISH(1)所完成的工作。在 FINISH()宏里,虚拟机获取下一条指令,并从指令中提取操作码号,根据该操作码号到指令解释程序查找表中得到相应的标签,然后跳转到该处理程序执行。其定义如代码清单 5.9 所示。

代码清单 5.9　dalvik/vm/mterp/out/InterpC-portable.cpp:FINISH(_offset)

```
#define FINISH(_offset) {                                   \
        ADJUST_PC(_offset);                                 \
        inst=FETCH(0);                                      \
        if(self->interpBreak.ctl.subMode){                  \
            dvmCheckBefore(pc,fp,self);                     \
        }                                                   \
        goto * handlerTable[INST_INST(inst)];               \
}
```

其主要流程有如下几步。

(1) FINISH 宏的参数是偏移地址,使用宏 ADJUST_PC(_offset)来调整 PC。ADJUST_PC 宏首先会判断偏移地址是否溢出,若是会记录错误停止虚拟机,否则会根据偏移地址来调整 PC 值。

(2) 接着由 FETCH 指令取指:

代码清单 5.10　dalvik/vm/mterp/out/InterpC-portable.cpp:FETCH(_offset)

```
#define FETCH(_offset)        (pc[(_offset)])
```

(3) 判断是否正常解释指令模式:self→interpBreak.ctl.subMode,若不是则调用 dvmCheckBefore(pc, fp, self)进行指令检查。一般情况下在调试或性能测试时,需要进行指令检查。

(4) 根据操作码使用 INST_INST 获取指令的索引号。

代码清单 5.11　dalvik/vm/mterp/out/InterpC-portable.cpp:INST_INST(_inst)

```
#define INST_INST(_inst)        ((_inst) & 0xff)
```

(5) 最后使用 goto * handlerTable[INST_INST(inst)]跳转到指令标签处开始执行。

以图 5.3 中指令的执行顺序为例,解释器从解释 NOP 指令开始,由于 NOP 指令不做任何事,于是马上开始取下一条指令。取得的指令为 MOVE 指令,处理后得到 MOVE 的索引号为 1,于是就得到 Goto Table 的下标为 1 的元素内容,该元素存储的是 MOVE 指令标签的地址。最后用 goto 语句跳转到该地址开始 MOVE 指令的解释。

5.3.3　一个解释程序的例子

在 Dalvik 中,有二百多条指令,每一条指令都有对应的解释程序。本节以整型的算术运算来分析 Portable 解释器中的指令解析程序是如何实现的。

在 Dex 文件中两个 int 型变量进行乘法运算的指令如下所示。其中,v1 和 v2 表示的是

寄存器。其对应的操作符和枚举值为 OP_MUL_INT。

```
mul-int/2addr v1,v2
```

通过字节码解析的原理知道，这一指令所对应的指令解析程序是经过宏替换后 op_OP_MUL_INT 标签对应的程序。在 InterpC-portable.Cpp 中可以找到其对应的解释程序为：

```
HANDLE_OP_X_INT(OP_MUL_INT,"mul",*,0)
OP_END
```

其中，HANDLE_OP_X_INT 仍是一个宏定义：

```
#define HANDLE_OP_X_INT(_opcode,_opname, _op,_chkdiv)      \
    HANDLE_OPCODE(_opcode /* vAA, vBB, vCC */){…}
```

替换后指定了其中_opcode 为 OP_MUL_INT，也即 HANDLE_OPCODE(_opcode)对应的是 op_ OP_MUL_INT。操作码名称_opname 为“mul”，操作_op 的值为“ * ”，除法标志位_chkdiv 为 0。对于整型加法 add-int、整型减法 sub-int、整型除法 div-int 的解释程序均有以下定义：

```
HANDLE_OP_X_INT(OP_ADD_INT,"add",+,0)
HANDLE_OP_X_INT(OP_SUB_INT,"sub",-,0)
HANDLE_OP_X_INT(OP_DIV_INT,"div",/,1)
```

可以看到，4 种运算使用同一个宏定义，通过传递不同的参数来进行不同的运算。接下来就详细分析 HANDLE_OPCODE(_opcode)的具体实现。

(1) 使用 INST_AA(inst)获取目的寄存器号。

```
vdst= INST_AA(inst)
```

(2) FETCH 进行取指，获取两个操作数 vsrc1 和 vsrc2。

```
Vsrc1= srcRegs & 0xff;
Vsrc2= srcRegs>> 8;
```

(3) 判断除法标志位_chkdiv，若值为 0，表示为加、减或乘法运算，则取得两个操作数的值进行对应的运算，并将结果存入寄存器 vdst 中：

```
SET_REGISTER(vdst,      \
        (s4) GET_REGISTER(vsrc1) _op (s4) GET_REGISTER(vsrc2));
```

(4) 若 _chkdiv 值为 1，进行除法运算，首先获得两个操作数的值 firstVal 和 secondVal，判断除数 secondVal 的值是否为 0，若是，则记录当前 PC 值并抛出“除以零”的算术运算异常，并跳转到异常处理部分。若不是，则接下来判断 firstVal 为 0 或者 secondVal 为－1 的情况，分别处理并将结果存入 result，若不符合上一个情况则直接使用 result＝firstVal _op secondVal 式子完成运算，最后使用 SET_REGISTER(vdst, result) 将结果存入寄存器。

(5) 运算完成使用 FINISH(2)宏取得下一条指令并执行。

其完整代码如代码清单 5.12 所示。

代码清单 5.12　dalvik/vm/mterp/out/InterpC-portable.cpp:HANDLE_OP_X_INT

```
#define HANDLE_OP_X_INT(_opcode, _opname, _op, _chkdiv)                    \
    HANDLE_OPCODE(_opcode /* vAA, vBB, vCC */)                              \
    {                                                                       \
        u2 srcRegs;                                                         \
        vdst=INST_AA(inst);                                                 \
        srcRegs=FETCH(1);                                                   \
        vsrc1=srcRegs & 0xff;                                               \
        vsrc2=srcRegs >> 8;                                                 \
        ILOGV("|%s-int v%d,v%d", (_opname), vdst, vsrc1);                   \
        if (_chkdiv != 0) {                                                 \
            s4 firstVal, secondVal, result;                                 \
            firstVal=GET_REGISTER(vsrc1);                                   \
            secondVal=GET_REGISTER(vsrc2);                                  \
            if (secondVal==0) {                                             \
                EXPORT_PC();                                                \
                dvmThrowArithmeticException("divide by zero");              \
                GOTO_exceptionThrown();                                     \
            }                                                               \
            if ((u4)firstVal==0x80000000 && secondVal==-1) {                \
                if (_chkdiv==1)                                             \
                    result=firstVal; /* division */                         \
                else                                                        \
                    result=0;        /* remainder */                        \
            } else {                                                        \
                result=firstVal _op secondVal;                              \
            }                                                               \
            SET_REGISTER(vdst, result);                                     \
        } else {                                                            \
            /* non-div/rem case */                                          \
            SET_REGISTER(vdst,                                              \
                (s4) GET_REGISTER(vsrc1) _op (s4) GET_REGISTER(vsrc2));     \
        }                                                                   \
    }                                                                       \
    FINISH(2)
```

流程图如图 5.5 所示。

Porable 解释器中一部分指令解释程序采取的是一个指令对应一段不同的解释程序，而另一部分如算术运算、if 语句等的解释程序均采取上述这种形式：将同一类型的解释程序使用一个宏定义实现，通过传递不同的参数来进行不同运算。

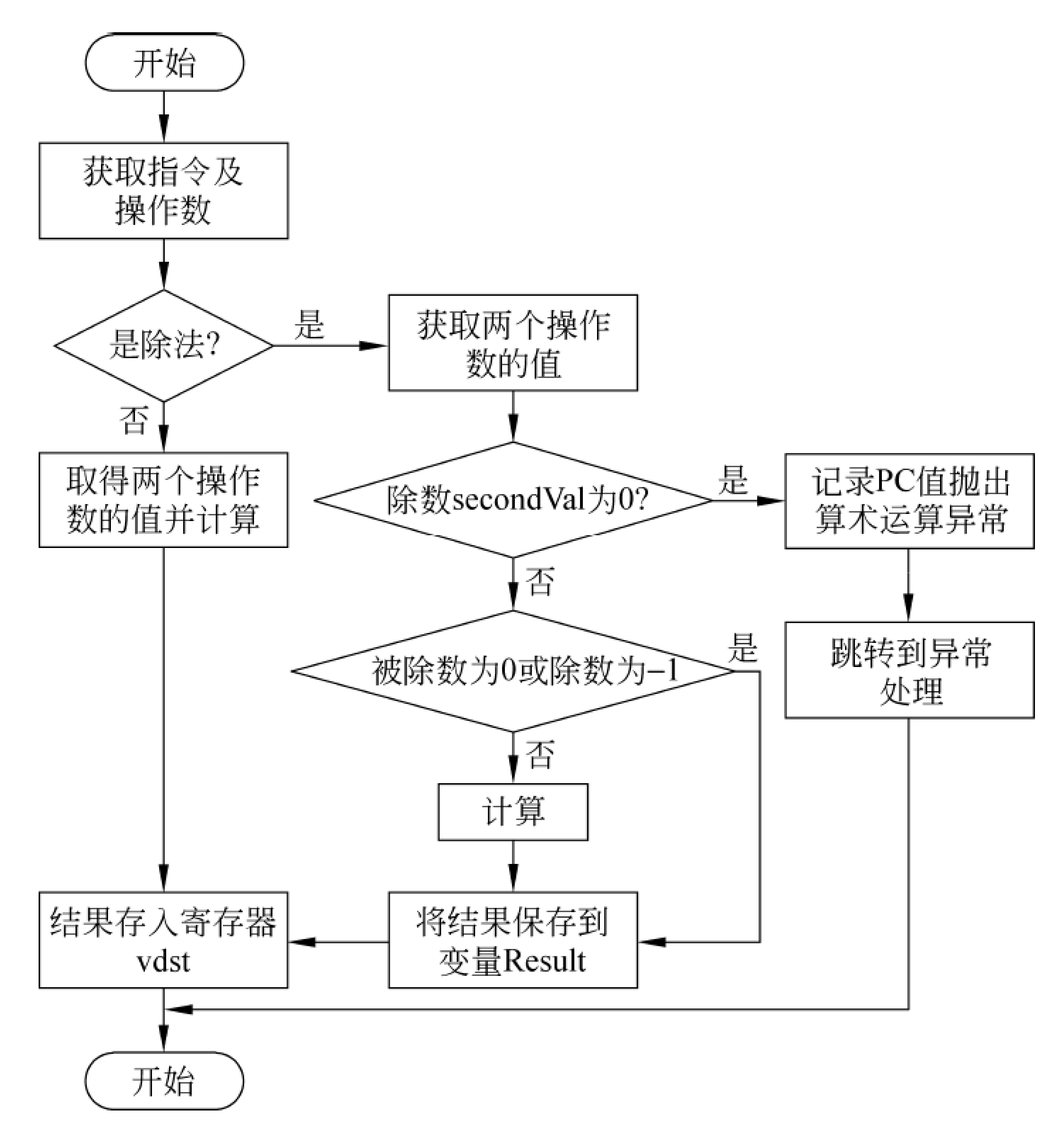


图 5.5 算术运算指令解释程序流程

5.4 Fast 解释器 C 实现分析

Fast,顾名思义,有更快的解释速度。Fast 解释器是汇编实现版本,那为何还要提供 C 实现版本呢?当前硬件平台众多,且发展十分迅速,为这些硬件平台及时提供一个针对硬件设计并由汇编实现的解释器也就比较困难。因此,提供了 C 实现版本,可以根据硬件平台状况,可以糅合用汇编实现的和由 C 实现的解释程序。因此,接下来从字节码解析原理和字节码指令解释流程两个方面来讲述 C 实现的 Fast 解释器。

5.4.1 字节码解析原理

同样作为 C 实现的解释器,Fast 解释器和 Portable 解释器有许多的代码复用。而这一切都是通过一系列的宏实现的。与 Portable 类似,在该版本的解释器中也使用 5.3.1 节中所述的静态数组,所不同的是数组中的元素是解释程序对应的函数指针。同样地也是利用 DEFINE_GOTO_TABLE 这个宏,所不同的是静态数组元素的宏 H 将被替换为 dvmMterp_##_op,如代码清单 5.13 所示。

代码清单 5.13 dalvik/vm/mterp/out/InterpC-allstubs.cpp

```
#undef H
#define H(_op) dvmMterp_##_op
DEFINE_GOTO_TABLE(gDvmMterpHandlers)
```

```
#undef H
#define H(_op) #_op
DEFINE_GOTO_TABLE(gDvmMterpHandlerNames)
```

结合 Portable 版本解释器实现，可以看到，这里定义了两个静态数组，数组中的元素是函数指针，函数名以 dvmMterp_开头。gDvmMterpHandlers 数组对应的是各个操作码解释程序的入口地址；gDvmMterpHandlerNames 对应的是各个操作码的名称，在输出日志时可以用到。于是，定义的两张表格如图 5.6 和图 5.7 所示。

0x00	dvmMterp_OP_NOP
0x01	dvmMterp_OP_MOVE
	…
0x59	dvmMterp_OP_IPUT
0x5a	dvmMterp_OP_IPUT_WIDE
	…

图 5.6　操作码解释程序函数指针

0x00	OP_NOP
0x01	OP_MOVE
	…
0x59	OP_IPUT
0x5a	OP_IPUT_WIDE
	…

图 5.7　操作码解释程序名称

在 C 实现的 Fast 解释器中，各个指令对应的程序不再以标签指明，而是以函数的形式存在。因此在建立的 gDvmMterpHandlers 中存储的就是这些函数的地址，如图 5.8 所示。

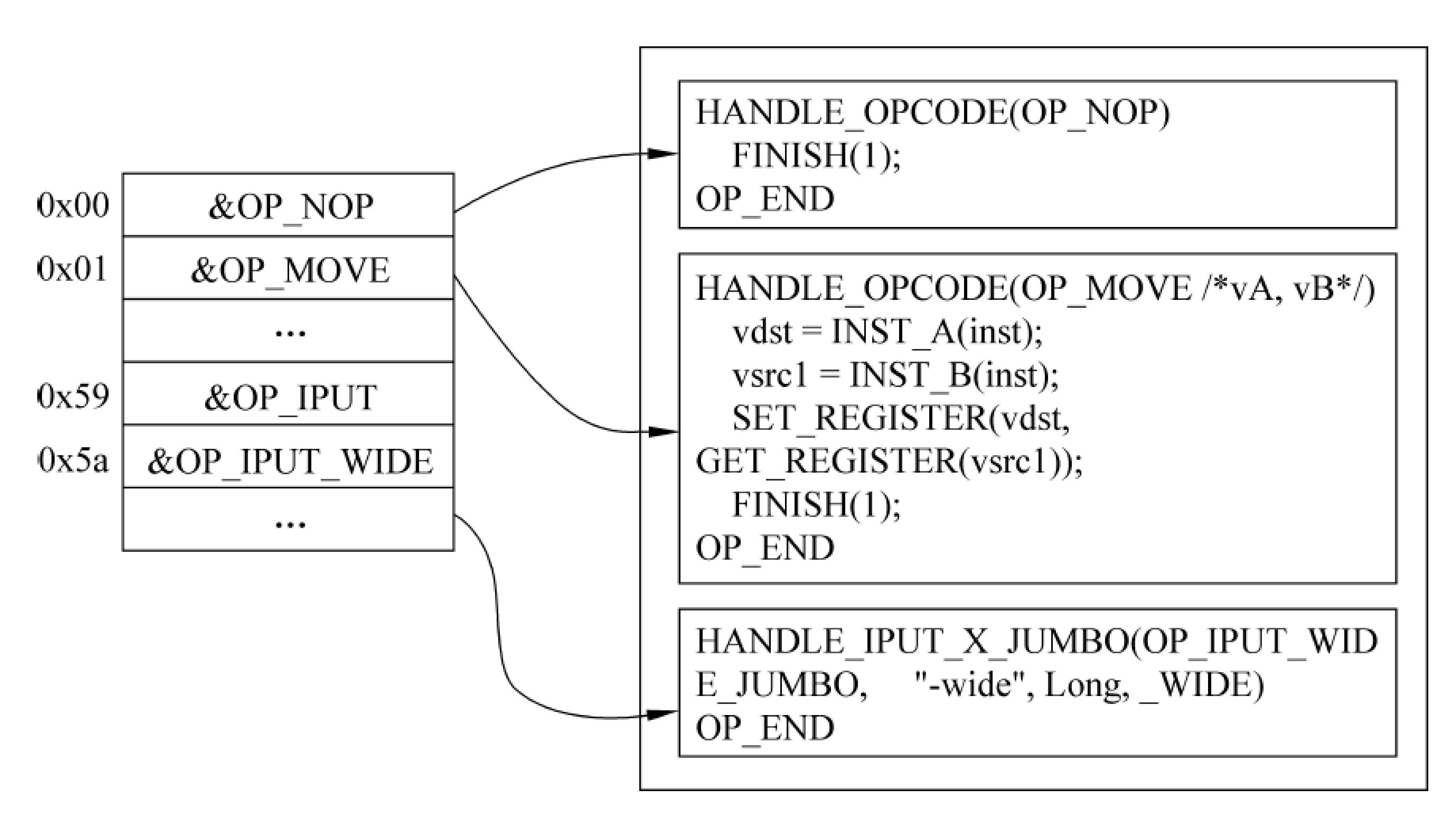


图 5.8　函数指针表

读者应该已经发现，其索引方式是和 Portable 一样的，都是以操作码类型对应的枚举值为数组下标，取得解释函数的函数指针，再进行解释。

5.4.2　字节码指令解释流程

Fast 解释器通过一个 while 循环来实现取指、指令检查、跳转到相应函数来解释指令、指令解释完成后回到主函数。在每个 while 循环开始，解释器取得将要解释的指令并进行指令检查，然后根据指令，得到函数指针表中的索引号，根据索引号得到函数地址并跳转执行函数，指令解释完成后又回到主函数，开始下一个指令的解释。既不同于普通的 switch 方式，也不同于 Gcc 的 Thread Coded 技术。

Fast 解释器入口函数是 dvmMterpStd，在这个函数中执行解释器环境的配置，通过调用 dvmMterpStdRun 函数来完成字节码解析。在 Fast 解释器的两种实现中分别使用 C 语言和汇编语言实现了 dvmMterpStdRun 函数。代码清单中所示的 while 循环在 dalvik/vm/mterp/out/InterpC-allstubs.cpp 文件中的 dvmMterpStdRun 函数中。while 循环的判断条件一直为真，当指令是长跳转时，才跳出这个循环。其执行流程图如图 5.9 所示。

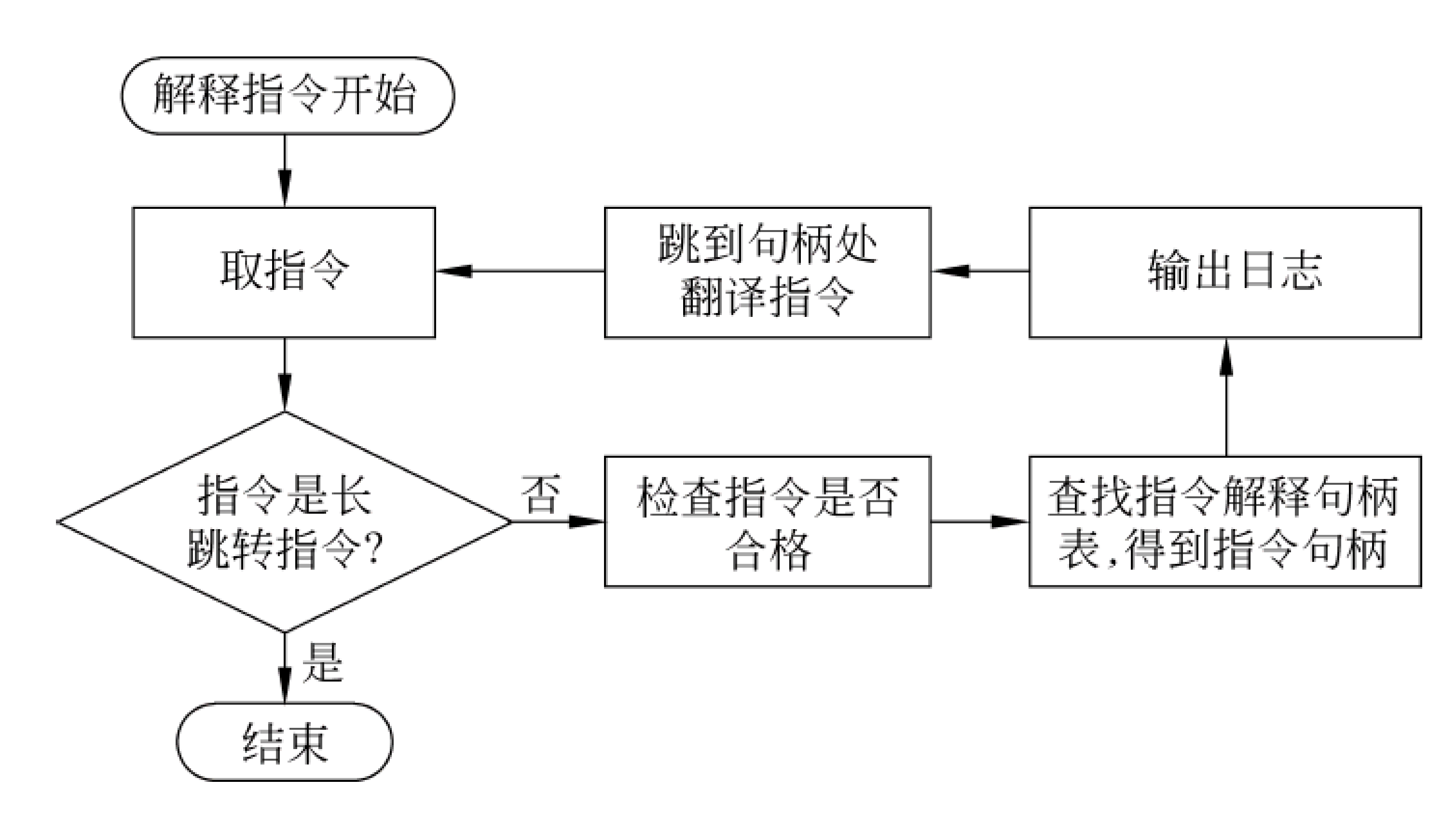


图 5.9 dvmMterpStdRun 函数执行流程图

在 while 循环的主体部分，主要有以下几点和 Portable 不同：①取指；②FINISH 宏；③解释程序的构造。

在伪代码中可以看到，取指在 while 开始后所要做的第一步，已经和 FINISH 宏割离开。和 Portable 在指令结束后直接取指跳转不同，需要先返回主函数。其具体如下所示，只要获得 pc 数组中的第一条指令即可。

```
u2 inst=/* self->interpSave.* /pc[0];
```

在 FINISH 宏中取消了取指操作后，FINISH 宏也只需要进行 pc 值的调整，不再需要做取指的动作。

最后，和 Portable 不同的是解释程序形式上都是函数。在 Fast 版的 C 语言实现中，函数的定义用的都是 c 文件夹下的代码片段，所以和 Portable 版的解释程序在形式上是一样的，只是用不同的宏来替换后，在 Fast 版中就成为函数，而在 Portable 版则为标签。操作码解释程序中每一个操作码都对应一个函数，函数的参数是 self 指针，并且返回空。在文件中同时定义了两个操作码解释程序代码片段的宏，如代码清单 5.14 所示。

代码清单 5.14 dalvik/vm/mterp/out/InterpC-allstubs.cpp:HANDLE_OPCODE(_op)

```
#define HANDLE_OPCODE(_op)                                           \
    extern "C" void dvmMterp_##_op(Thread* self);                    \
    void dvmMterp_##_op(Thread* self) {                              \
        u4 ref;                                                      \
        u2 vsrc1,vsrc2,vdst;                                         \
        u2 inst=FETCH(0);                                            \
        (void)ref; (void)vsrc1; (void)vsrc2; (void)vdst; (void)inst;
#define OP_END    }
```

以 NOP 操作为例，其实现代码如代码清单 5.15 所示。可以看到 NOP 的具体操作和

Portable 实现没有本质的差别。

代码清单 5.15　dalvik/vm/mterp/out/InterpC-allstubs. cpp

```
HANDLE_OPCODE(OP_NOP)
    FINISH(1);
OP_END
```

在对上述两个不同定义的宏预处理后，就定义了一个函数。最终实现的代码如代码清单 5.16 所示。

代码清单 5.16　dalvik/vm/mterp/out/InterpC-allstubs. cpp

```
extern "C" void dvmMterp_OP_NOP(Thread* self);
void dvmMterp_OP_NOP(Thread* self) {
    u4 ref;
    u2 vsrc1,vsrc2,vdst;
    u2 inst=FETCH(0);
(void)ref; (void)vsrc1; (void)vsrc2; (void)vdst; (void)inst;
FINISH(1);
}
```

所有的操作码都像 NOP 操作那样定义，经过预处理后得到了操作指令函数的定义。函数指针表中的元素是对应的解释程序的函数指针。每段指令解释程序则是对应函数的具体实现，这样建立起函数指针表和解释程序一一对应的关系。通过查找函数指针表，得到函数指针，即可跳转到解释程序函数的实现位置执行。

点拨　阅读 Dalvik 解释器的代码，由衷感到宏的强大，以及因此带来的代码复用的魅力。C 实现的 Fast 解释器相比于 Portable 解释器，更改的代码非常少。

5.5　Fast 解释器汇编实现分析

汇编实现的解释器是针对平台优化的。在 vm/interp/out 目录下可以看到依据不同平台而命名的 InterpC-<arch>. c，InterpAsm-<arch>. S 代码，可以针对不同的平台采取不同的指令集。在 Dalvik 中，主要有 ARM 和 x86 两种平台的实现。下面来看下针对两个平台的实现有何不同。其中，ARM 平台主要分析 InterpAsm-Armv7-a-neon. S，x86 平台主要分析 InterpAsm-x86. S。

5.5.1　字节码解析原理

和 Portable 的 Threaded Code 机制类似，Fast 解释器采用了 computed-goto 和 jump-table 的机制实现自动取指。ARM 采用前者，x86 采用后者。

所谓的 computed-goto 机制，实现上，各个指令的解释程序严格限制在固定大小。ARM 平台默认为 64B。在内存中，按照字节码号的顺序从名为 dvmAsmInstructionStart 的地址处开始，依次紧挨着排列。如果某条指令解释程序大小小于 64B，剩余空间就空着；如果大于 64B，则这条指令应该放在附加空间里，每条指令处理程序以适当的方式在主处理程序空间和附加空间之间建立跳转关系。如代码清单 5.17 所示，以 L_OP_NOP 开始，依

次以.balign 64 对齐 64B。在获取指令类型枚举值后，根据公式 dvmAsmInstructionStart＋OP ×64 即可跳转到下一条指令。

代码清单 5.17 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
dvmAsmInstructionStart= .L_OP_NOP
    .text
    .balign 64
.L_OP_NOP: /* 0x00 */
    FETCH_ADVANCE_INST(1)            @ advance to next instr,load rINST
    GET_INST_OPCODE(ip)              @ ip<-opcode from rINST
    GOTO_OPCODE(ip)                  @ execute it
    .balign 64
.L_OP_MOVE: /* 0x01 */
    /* for move,move-object,long-to-int */
    /* op vA,vB */
    mov      r1,rINST,lsr #12        @ r1<-B from 15:12
    ubfx     r0,rINST,#8,#4          @ r0<-A from 11:8
    FETCH_ADVANCE_INST(1)            @ advance rPC,load rINST
    GET_VREG(r2,r1)                  @ r2<-fp[B]
    GET_INST_OPCODE(ip)              @ ip<-opcode from rINST
    SET_VREG(r2,r0)                  @ fp[A]<-r2
    GOTO_OPCODE(ip)                  @ execute next instruction
    .balign 64
.L_OP_MOVE_FROM16: /* 0x02 */
```

以图形展示，具体的指令安排如图 5.10 所示。

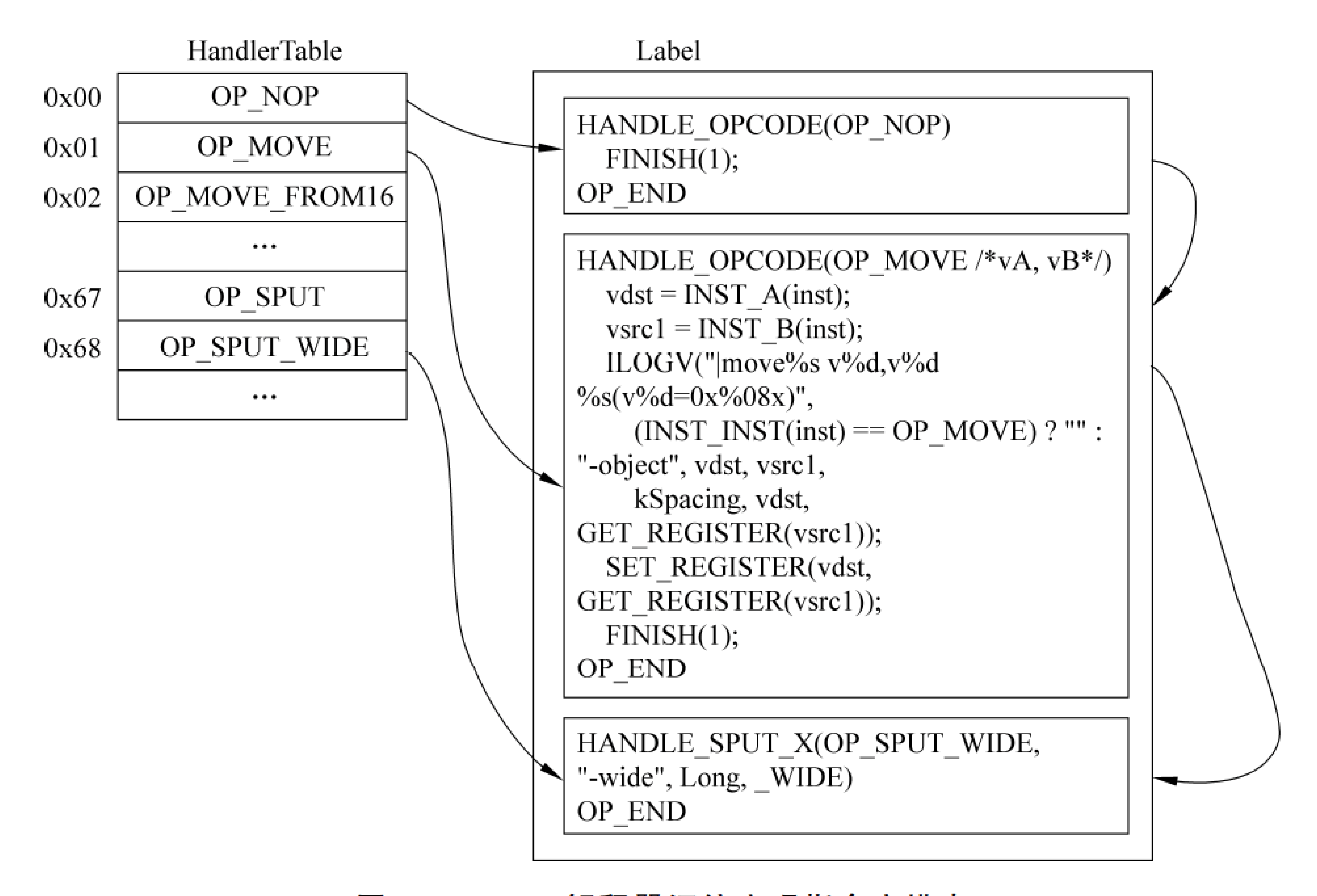


图 5.10 Fast 解释器汇编实现指令安排表

另一方面，所谓 jump-table 机制，是指和 Portable 解释器类似，也通过一个数组来确定接下来跳转的地址。在 x86 平台的配置文件中可以看到，x86 正是采用的这种方式。与 ARM 不同，所有的指令解释程序存储在连续的地址中，每个指令解释程序所占的内存大小是任意的。在以 dvmAsmInstructionStart 地址开始处存储了指令解释程序的地址，可根据操作码跳转到相应的解释程序。如代码清单所示，其定义和 ARM 类似，只是少了 64B 的对齐。但是定义了如代码清单 5.18 所示的 jump-table，其中存储的是各个指令的标签，和 Portable 解释器中的静态数组使用方式类似，根据首地址 dvmAsmInstructionStart 和指令类型枚举值得到其跳转的 Label。

代码清单 5.18　dalvik/vm/mterp/out/InterpAsm-x86.S

```
dvmAsmInstructionStartCode= .L_OP_NOP
    .text

.L_OP_NOP: /* 0x00 */
/* File: x86/OP_NOP.S */
    FETCH_INST_OPCODE 1 %ecx
    ADVANCE_PC 1
    GOTO_NEXT_R %ecx

.L_OP_MOVE: /* 0x01 */
/* File: x86/OP_MOVE.S */
    /* for move,move-object,long-to-int */
    /* op vA,vB */
...

.L_OP_MOVE_FROM16: /* 0x02 */
/* File: x86/OP_MOVE_FROM16.S */

    .global dvmAsmInstructionStart
    .text
dvmAsmInstructionStart:
    .long .L_OP_NOP /* 0x00 */
    .long .L_OP_MOVE /* 0x01 */
    .long .L_OP_MOVE_FROM16 /* 0x02 */
    ...
```

指令解释程序的内存构造以及跳转执行方式如图 5.11 所示。

5.5.2　字节码解析流程

在当前版本由汇编实现的解释器中，运作原理和 Portable 解释器类似。在解释程序结束后，直接取指、查找表并跳转。解释器主线程通过调用 dvmMterpStd 函数使用 Fast 解释器进行解析，在该函数中通过调用 dvmMterpStdRun 函数来实现字节码解析。

在解释器从 dvmMterpStd 函数的 C 代码转入汇编实现的 dvmMterpStdRun 函数之前，需要定义一个结构体，用于保存当前状态，如将要执行的方法、指令的入口、堆栈信息等，

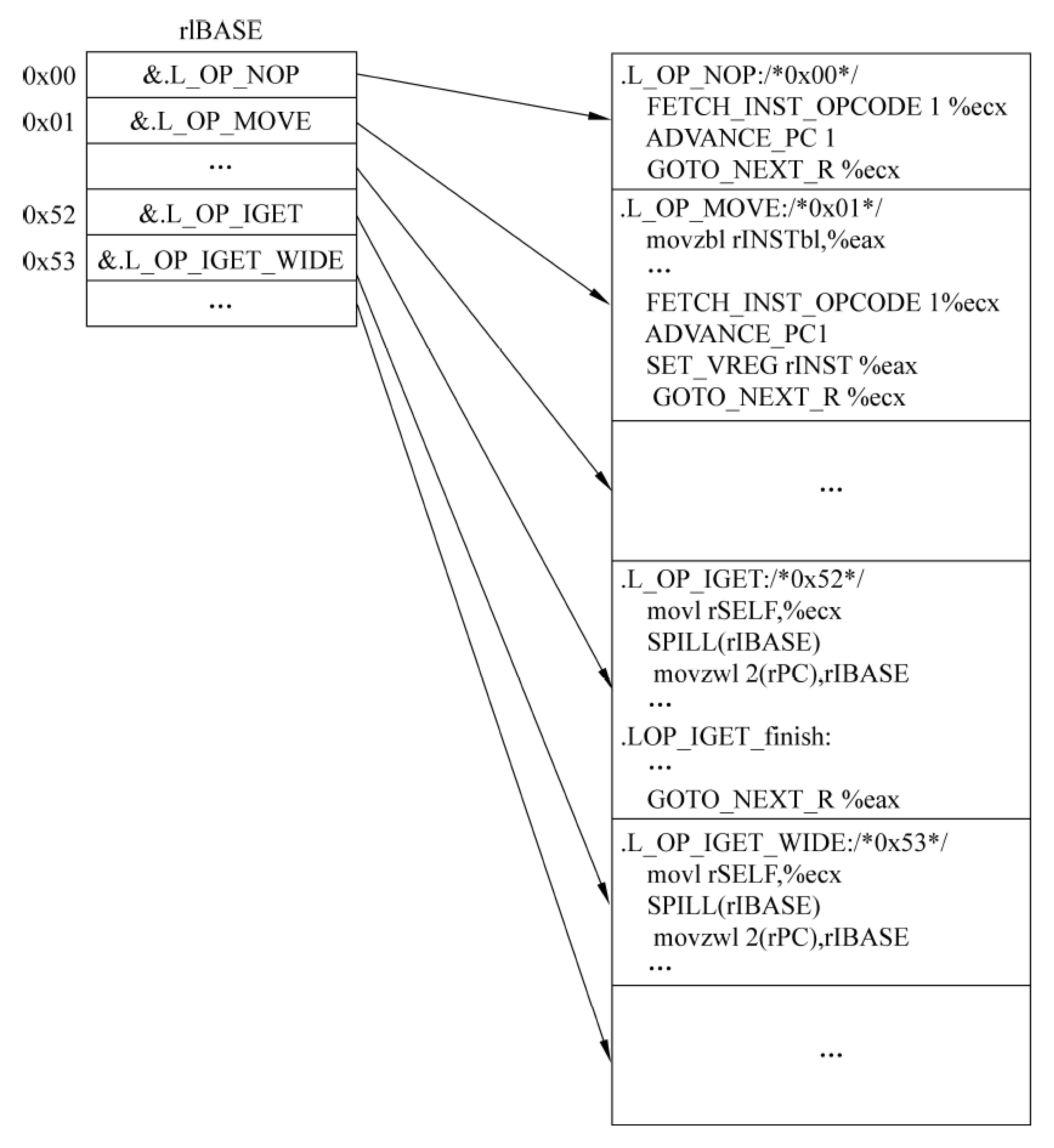


图 5.11　x86 汇编实现指令解释程序的内存构造图

如代码清单 5.19 所示。

代码清单 5.19　dalvik/vm/interp/InterpSate.h

```
struct InterpSaveState {
    const u2*       pc;             /**Dalvik 虚拟机 PC,当前将要执行指令 */
    u4*             curFrame;       /**Dalvik 虚拟机帧指针 */
    const Method    * method;       /**开始要被执行的方法 */
    DvmDex*         methodClassDex; /**类中方法的字节码 */
    JValue          retval;         /**返回值,可能是数值类型或对象 */
    void*           bailPtr;        /**保持指针 */
    ...
};
```

在进入汇编代码之前会保存将要解释的类的方法的字节码信息,如代码清单 5.20 所示。

代码清单 5.20　dalvik/vm/interp/Interp.cpp:dvmInterpret

```
self->interpSave.method=method;
self->interpSave.curFrame=(u4*)self->interpSave.curFrame;
self->interpSave.pc=method->insns;
```

```
* pResult= self- > interpSave.retval;
/* Restore interpreter state from previous activation * /
self- > interpSave= interpSaveState;
```

根据不同的平台实现，执行相应的 dvmMterpStrRun 函数。接下来分别针对 ARM 平台和 x86 平台讲解。

1. ARM 平台

针对 ARM 平台，Dalvik 虚拟机启动时，需要检查指令的解释程序是否超过大小，若有指令超过大小，虚拟机将报错并因此退出。需要注意的是，出错后并不提示哪条指令的解释程序出错了。

和 Portable 类似，在进入指令解释前，需要保存现场。在汇编版的 Fast 解释器中则表现为对涉及的寄存器的保存。代码如代码清单 5.21 所示。

代码清单 5.21 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
#define MTERP_ENTRY1 \
    .save {r4- r10,fp,lr}; \
    stmfd sp!,{r4- r10,fp,lr}          @ save 9 regs
#define MTERP_ENTRY2 \
    .pad  #4; \
    sub   sp,sp,#4                     @ align 64

    .fnstart
    MTERP_ENTRY1
    MTERP_ENTRY2

    /* save stack pointer,add magic word for debuggerd * /
    str   sp,[r0,#offThread_bailPtr]   @ save SP for eventual return
```

在保存完现场后，则开始指令解释，流程图如图 5.12 所示。

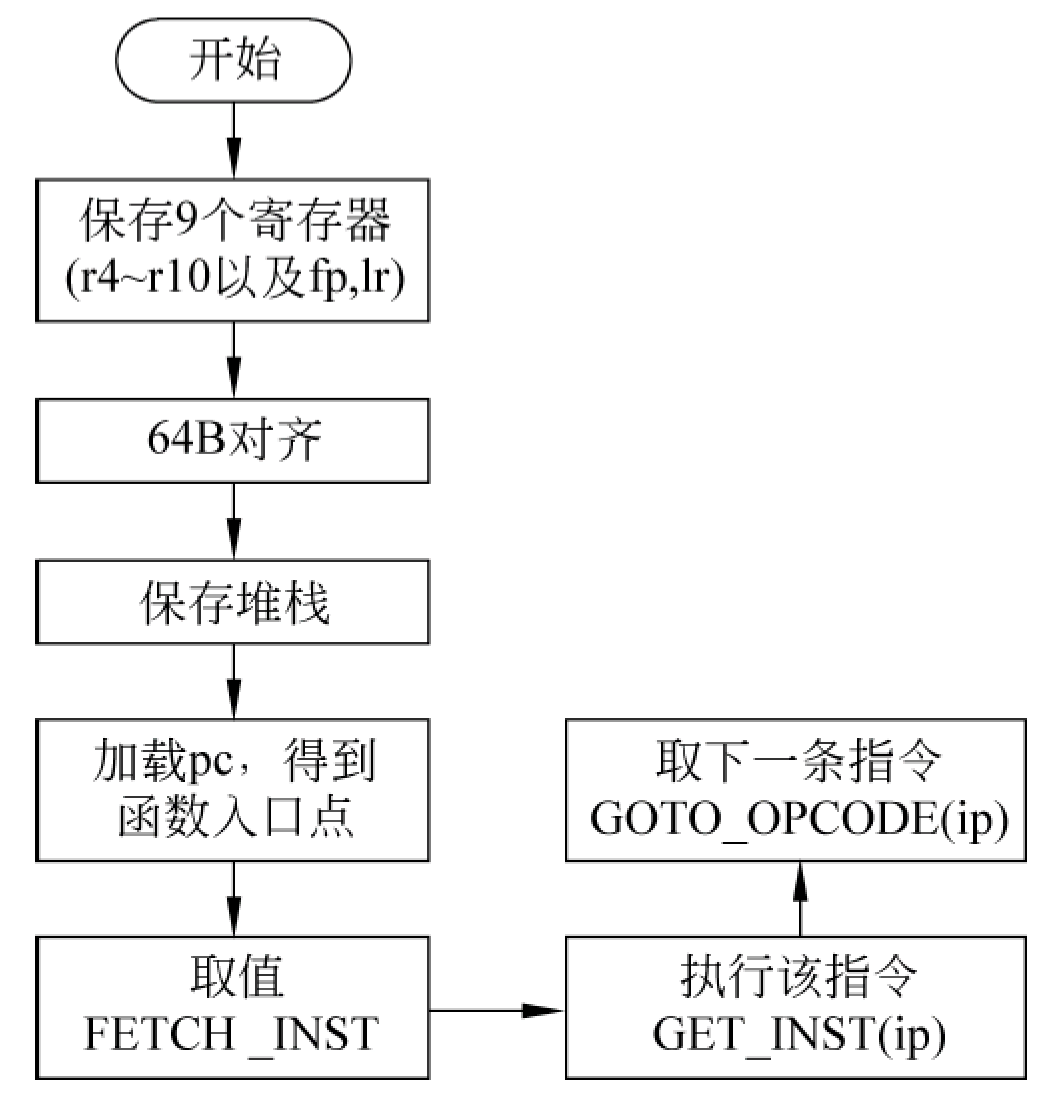


图 5.12 Fast 解释器汇编实现执行流程

汇编版本实现和 All-stubs 版本实现基本一样，最大的差异在于汇编版本不是用一个大的 while 循环来控制指令的解释，而是以以下三条语句开始解释器的执行。

代码清单 5.22 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
FETCH_INST()              @  load rINST from rPC
GET_INST_OPCODE(ip)       @  extract opcode from rINST
GOTO_OPCODE(ip)           @  jump to next instruction
```

FETCH_INST 从 rPC 寄存器中获取指令，是一个宏定义，定义如代码清单 5.23 所示。

代码清单 5.23 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
#define FETCH_INST()        ldrh   rINST,[rPC]
```

Ldrh 是 ARM 指令，rPC 指向的是字节码存储地址，将该地址中的内容加载到 rINST 寄存器中。之后用 GET_INST_OPCODE 开始获取指令类型枚举值，其具体实现也为一个宏定义。

代码清单 5.24 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
#define GET_INST_OPCODE(_reg)          and      _reg,rINST,#255
```

And 是 ARM 指令，获取存储在 rINST 寄存器中的字节码的操作码号，并放置在_reg 寄存器中，_reg 将被替换为 ip。最后跳转到该字节码对应的解释程序处开始解释。

正如前文所说，ARM 中采用的是 computed-goto 机制，每块解释程序都被放置在 64B 的内存中，不满空着，超出则另加。GOTO_OPCODE 需要根据以下公式计算得到指令解释程序地址。

```
dvmAsmInstructionStart+OP×64
```

反映在代码上则如代码清单 5.25 所示。

代码清单 5.25 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
#define GOTO_OPCODE(_reg)              add      pc,rIBASE,_reg,lsl #6
```

在每条指令执行结束后，依然是取值、跳转、执行三个步骤，实现上和上面略有差别，可以看到和 Portable 标签有点类似，也是在每条指令解释程序完成前进行取指并跳转。直至解释结束。

代码清单 5.26 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
FETCH_ADVANCE_INST(1)          @advance rPC,load rINST
GET_VREG(r2,r1)                @r2<-fp[B]
GET_INST_OPCODE(ip)            @ip<-opcode from rINST
```

2. x86 平台

从函数 dvmInterp 进入面向 x86 平台的 Fast 解释器的汇编实现的 dvmMterpStdRun 函数开始解析，其字节码解析流程如图 5.13 所示。

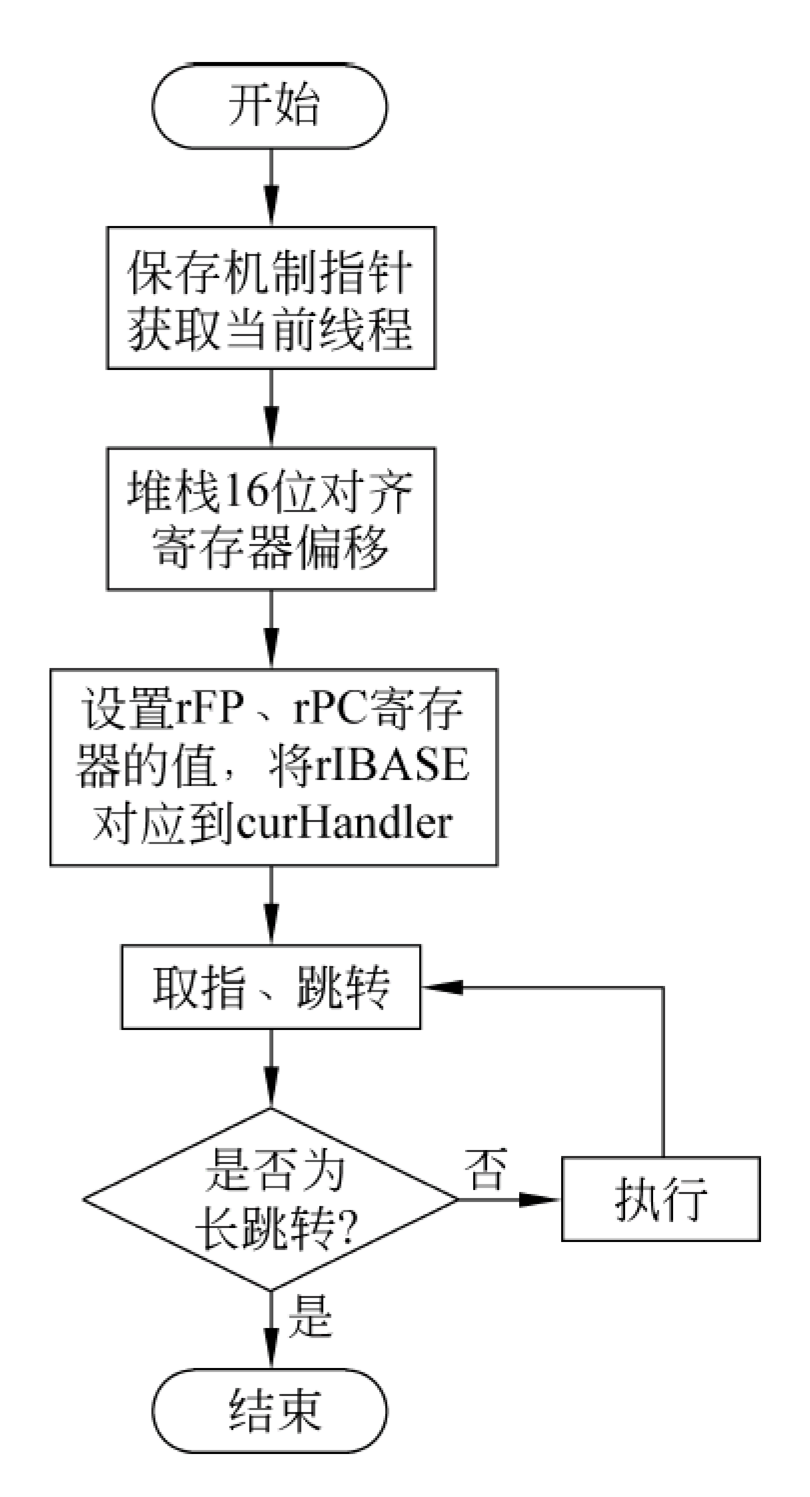


图 5.13　面向 x86 平台指令解析流程图

从流程图中可看到，在取指解释之前需要对特定功能的寄存器初值进行设置，如 rPC 是程序计数器，使用 offThread_pc 设置，其定义如下。

代码清单 5.27　dalvik/vm/mterp/out/InterpAsm-x86.S

```
MTERP_OFFSET(offThread_pc,Thread, interpSave.pc, 0)
```

最终这个宏定义会生成一个常量标签即：offThread_pc＝0。类似的还有栈帧 rFP、堆栈基址 rIBASE。

代码清单 5.28　dalvik/vm/mterp/out/InterpAsm-x86.S

```
MTERP_OFFSET(offThread_curFrame,Thread, interpSave.curFrame,4)
MTERP_OFFSET(offThread_curHandlerTable,Thread,interpBreak.ctl.curHandlerTable,44)
```

这里需注意的是，curHandlerTable 结构体 interpBreak 的成员，这一结构体用于解释程序的控制，其包含解释器模式信息，即线程被挂起的次数，当降到 0 时，线程重启。而数组 curHandlerTable 则指示当前的执行表。

程序准备完毕后，使用以下两条语句开始解释器的执行。

(1) FETCH_INST：和 ARM 平台类似，只是宏定义不再相同。从程序计数器 rPC 中获取下一条指令存入 rINST 中，并不调整 rPC，该宏定义如下。

代码清单 5.29　dalvik/vm/mterp/out/InterpAsm-x86.S

```
macro FETCH_INST
    movzwl      (rPC),rINST
```

(2) GOTO_NEXT：获取操作码并根据操作码进行跳转执行，该宏定义如下。

代码清单 5.30 dalvik/vm/mterp/out/InterpAsm-x86.S

```
.macro GOTO_NEXT
     movzx   rINSTbl,%eax
     movzbl  rINSTbh,rINST
     jmp      * (rIBASE,%eax,4)
.endm
```

在 GOTO_NEXT 宏的定义中，首先获取指令的操作码 rINSTbl 存入%eax 寄存器，之后获取指令的操作数 rINSTbh，通过 jmp 跳转指令跳转到 rIBASE+%eax×4 的地址开始执行解析。

在每条指令执行结束后，依然是取值、跳转执行，实现上和上面略有差别，可以看到和 Portable 标签有点类似，也是在每条指令解释程序完成前进行取指跳转。

（1）FETCH_INST_OPCODE 获取操作码存入指定寄存器。

代码清单 5.31 dalvik/vm/mterp/out/InterpAsm-x86.S

```
.macro FETCH_INST_OPCODE _count _reg
    movzbl   \_count* 2(rPC),\_reg
```

FETCH_INST_OPCODE 获取操作码并存入_reg，这个指令必须与 GOTO_NEXT_R _reg 一起使用。

（2）GOTO_NEXT_R 实现指令的跳转，定义如下。

代码清单 5.32 dalvik/vm/mterp/out/InterpAsm-x86.S

```
.macro GOTO_NEXT_R _reg
     movzbl   1(rPC),rINST
     jmp       * (rIBASE,\_reg,4)
.endm
```

最后跳转到 rIBASE+_reg×4 处执行。

指令跳转采用的是 jump-table。指令解释程序的地址从 dvmAsmInstructionStart 地址处按如下方式排列，每个指令解释程序的地址占 4B，并且以 rIBASE 作为指令解释程序的基址，因而，对于操作码为%eax 的指令其对应解释程序的地址为：rIBASE+%eax×4。

5.5.3 一个解释程序的例子

在 x86 平台下以算术运算为例，在 vm/mterp/out 文件夹下的 x86 平台实现 InterpAsm-x86.S 文件中可以找到，对于算术加、减、乘、除的简单运算指令解释程序分别进行了单独的实现，与 Portable 解释器的算术运算解释程序调用同一个宏不同，每个运算单独的指令解释程序减少了解释程序的判断复杂的判断逻辑，实现简单，其中乘法代码如下。

代码清单 5.33 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
.L_OP_MUL_INT: /*  0x92  * /
    movzbl   2(rPC),%eax                    #取得第一个操作数
    movzbl   3(rPC),%ecx                    #取得第二个操作数
    GET_VREG_R %eax %eax                    #取得操作数%eax的值
```

```
    SPILL(rIBASE)                      #设置基址 rIBASE
    imull   (rFP,%ecx,4),%eax          #计算操作
    UNSPILL(rIBASE)                    #重置基址 rIBASE
    FETCH_INST_OPCODE 2 %ecx           #获取下一操作码
    ADVANCE_PC 2                       #调整 PC
    SET_VREG %eax rINST                #%eax 存储结果
    GOTO_NEXT_R %ecx                   #跳转下一条指令
```

从代码中可以看到，首先获取操作数存入寄存器%eax 和%ecx 中，得到%eax 寄存器中的值，调整基址偏移量，使用 imull 指令进行乘法运算，重新调整基址，使用 FETCH_INST_OPCODE 获取下一条指令，ADVANCE_PC 调整程序 PC，SET_VREG 将计算结果存入指令寄存器中，GOTO_NEXT_R 跳转到下一条指令执行。对于加、减运算，只需将其中的 imull 指令替换为 addl 和 subl 指令即可，而对于除法指令，则需进行一些判断，这一过程与 Portable 解释器的对除法进行解释的流程相似，故不再详述。

在 InterpAsm-x86.S 文件中的指令解释程序与此类似。有些指令解释程序会涉及一些错误处理的情况，如在除法中，如果除数为 0，则会报错，用 common_errDivideByZero 错误处理，其定义如下。

代码清单 5.34　dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
common_errDivideByZero:
    EXPORT_PC
    movl    $.LstrDivideByZero,%eax
    movl    %eax,OUT_ARG0(%esp)
    call    dvmThrowArithmeticException
    jmp     common_exceptionThrown
```

在其中会调用到 C 代码编写的函数 dvmThrowArithmeticException，用来抛出异常。这与 Portable 解释器一致。

5.6　解释器的模块化设计

解释器采用了模块化的方法，允许开发特定于平台的代码，使得汇编语言编写的解释器更容易移植和调试。每个指令解释程序都是单独的文件，包括解释器的入口函数、出口函数以及宏定义，可以根据要求编写配置文件，从而得到相应平台的目标代码。

vm/mterp 文件夹包含 Dalvik 解释器的实现代码，包括解释器的 C 语言实现和汇编实现。在该文件夹下每个平台对应一个单独文件夹，文件夹中是这个平台相应操作的代码片段，包括实现 Portable 解释器的 C 代码文件，实现 Fast 解释器中针对 ARM 和 x86 平台的汇编代码文件。官方的源代码中已经有以下几种平台的汇编实现：①ARM v5te；②ARM v6；③ARM v6t2；④ARM v7-a；⑤ARM-vfp；⑥x86。不在以上几种平台的，可以用 C 语言实现的解释器，该版本解释器是对所有平台都通用的。

在解释器中使用配置文件来控制按照一定的规则组织 C 代码片段或汇编代码片段以生成对应类型或平台的解释器代码。在 mterp 目录下包括针对提及平台和类型的解释器配

置文件，都以“config-”开头，源代码被写入 out 文件夹目录下。

接下来说明配置文件的格式。配置文件自顶向下解析，在文件中每一行只有三种可能：①空行；②注释；③命令。配置文件要求可以填写以下几种命令中的全部或部分。

1. handler-style<computed-goto|jump-table|all-c>

该参数用来控制解释器的执行方式，分别是计算得到操作码解释程序地址(computed-goto)、采用跳转表跳转到操作码解释程序(jump-table)以及采用 C 函数方式来实现(all-c)。computed-goto 中，所有的操作码解释程序都被分配到固定的地址，每段解释程序都要小于一定大小，一般采用的是 64B。在解析时，通过计算 table-start-addres(opcode * handler-size)来得到解释程序地址。jump-table 则是根据操作码跳转到相应程序的解释程序，所以解释程序可以是任意大小的。all-c 参数则是为 Portable 解释器准备的。需要注意的是，该命令是必要的，且必须是配置文件的第一条命令。

2. handler-size<bytes>

该命令提供了一个值，以 bytes 为单位。汇编实现的解释程序占用的大小。在大部分的平台上，该值应该是 2 的幂。需注意，该命令是与 computed-goto 类型搭配使用的，对于 jump-table 和 all-c 实现是没有用的。

3. import<filename>

指定的文件中所有的内容将会拷贝到输出文件中。以“. cpp”和“. h”为后缀的文件将会拷贝到 C 文件中，以“. S”为后缀则会拷贝到汇编文件中。

4. asm-stub<filename>

该命令将会拷贝给定名字的文件，写入到输出文件时，会将操作码名称进行替换。主要完成的是汇编实现调用 C 语言函数。注意该命令不适用 all-c 实现。

5. asm-alt-stub<filename>

对于 computed-goto 实现，将会生成一系列的入口点，对于 jump-table 实现，则会生成一张可选择的跳转表。

6. op-start<directory>

表示开始生成操作码解释程序。根据配置文件指定的目标平台找到当前目录下相应的文件夹，该文件夹存储了一系列以“op”开头的文件，分别对应各个解释程序的实现。

7. op<opcode><directory>

将会用指定路径下的文件替换掉默认文件，相应的代码片段也将被替换掉。该参数必须在 op-start 命令后，在 op-end 命令前。

8. alt<opcode><directory>

和前述 op 命令一样，只不过替换的是选择表。

9. op-end

操作码列表结束。所有的操作码的解释程序都已经写到输出文件。任何其他不满足空间限制的程序(64B)就跟在操作码列表后。

对于每个特定的目标平台均有一个对应的配置文件，控制生成两个输出文件 InterpC-<arch>.C 和 InterpAsm<arch>.S。配置文件经由 gen-mterp.py 脚本进行解析，生成相应的解释器代码。依赖 Python 脚本文件处理功能，使得解释器的移植难度大大降低了。该脚本的用法是：

```
Python gen-mterp.py [平台版本][输出目录]
```

接下来分析 gen-mterp.py 脚本的实现原理。

首先通过获取系统参数得到目标平台的版本 target_arch 及输出文件目录 output_dir，获取操作码 opcodes 列表，打开目标平台的配置文件、以写的方式打开输出文件夹下的所应生成的针对目标平台的文件(包括.cpp 和.s 文件)。

```
config_fp=open("config-%s" %target_arch)
c_fp=open("%s/InterpC-%s.cpp" %(output_dir,target_arch),"w")
asm_fp=open("%s/InterpAsm-%s.S" %(output_dir,target_arch),"w")
```

以上准备工作完成后，开始生成目标平台的代码。生成输出文件的头部注释部分，得到配置文件的指针，读取配置文件的内容，空行和注释行跳过，判断命令是否为："handler-size"、"import"、"asm-stub"、"asm-alt-stub"、"op-start"、"op-end"、"alt"、"op"、"handler-style"字符串，对应调用 setHandlerSize(tokens)、importFile(tokens)、setAsmStub(tokens)、setAsmAltStub(tokens)、opStart(tokens)、opEnd(tokens)、altEntry(tokens)、opEntry(tokens)、setHandlerStyle(tokens)函数来执行相应字符串的功能。这些函数在脚本中已定义，用来实现相应的功能。若判断的命令不包含在以上字符串中，则报出命令错误的异常。最后关闭文件指针，代码生成完毕。

通过 gen-mterp.py 脚本会生成特定平台的.S 文件和.C 文件。

系统通过运行 rebuild.sh 会生成所有目标平台的代码：

```
for arch in portable allstubs armv5te armv5te-vfp armv7-a armv7-a-neon x86 x86-atom;do TARGET_ARCH_EXT
=$arch make-f Makefile-mterp;done
```

代码中通过 Makefile-mterp 文件来完成代码的生成，在 Makefile-mterp 文件中指定目标平台和输出目录，通过调用 gen-mterp.py 脚本完成生成：

```
/gen-mterp.py$(TARGET_ARCH_EXT)$(OUTPUT_DIR)
```

由于 Portable 解释器只针对 C 实现，因此其生成的汇编文件没有意义，在 rebuild.sh 脚本中将其除去：

```
rm -f out/InterpAsm-portable.S
```

由于解释器的模块化设计，若要生成针对特定平台的代码，只需要更改相关的参数或增加部分代码片段即可通过上述的方法生成新的平台代码，而不必修改庞大的生成代码。这对解释器在不同平台上的移植提供了方便。

小　　结

本章分别讲解了 Dalvik 的三种解释器实现，Portable、C 实现 Fast 和汇编实现 Fast。就这三种解释器实现的不同和使用场景的不同进行了深入阐述。最后详细说明了如何通过脚本实现了解释器的模块化设计。

第 6 章 即时编译模块的原理及实现

本章主要内容

- ✍ 什么叫 JIT？
- ✍ JIT 有哪些类型？
- ✍ Dalvik JIT 的整体工作流程是怎样的？
- ✍ Dalvik JIT 是如何和解释器耦合的？
- ✍ Dalvik JIT 的前后端是如何衔接的？
- ✍ Dalvik JIT 的前端部分是如何工作的？
- ✍ Dalvik JIT 的后端部分是如何工作的？

众所周知，程序执行有两种方式，分别为解释和编译。解释方式是逐句读取源程序逐句翻译成机器码再执行；而编译方式则是在运行程序前，将整个程序翻译为等价的目标程序，计算机直接执行该目标程序。JIT 混合了两种技术，解释器解释时，编译部分程序，并在下次直接执行该编译后的源程序。

6.1 概述

JIT(Just-In-Time)，中文含义为即时编译，又称为动态编译，执行时动态地编译程序，以缓解解释器的低效工作。对于 Java 这类语言来说，经过编译后，将生成和平台无关的中间代码。不同平台上的虚拟机(JVM)都可以执行相同的 Java 中间代码，同时需要处理和操作系统和硬件平台相关的部分。这也是大多数解释型语言采用的模型。虚拟机中最主要的是解释器，负责解释执行中间代码。

虽有跨平台的优势，但同时也带了运行效率低的直观感受。究其原因，是因为其执行原理是一句一句翻译字节码。比如，字节码中出现循环，根据解释器的原理，它需要重复地解释并执行这一组程序。这使得纯粹基于解释器运行的程序效率十分低下，造成了浪费。

JIT 是解决这个缺陷的一种有效手段。通过将字节码编译为 Native Code，让解释器不再重复执行这些热点代码片段。而且，相比于解释器，JIT 编译器可以更高效地利用 CPU 和寄存器。同时在编译的过程中，可以进行部分低级代码优化，比如常数传播、取消范围检查、复制传播等。尽可能地生成媲美编译器编译的二进制代码。之后执行编译生成的 Native Code，从而达到加速执行应用程序的目的。

但需要注意的是，JIT 也不是万能的。引入 JIT 将会消耗额外的时间和空间。相比于

正常的解释执行，如果 JIT 编译消耗的时间并没有占有优势，引入 JIT 不仅不会提升程序执行性能，反而会降低。

点拨 一些现代的解释器一般都包含 JIT 模块。其实，在互联网如此发达的现在，我们几乎无时无刻地在接触 JIT，因为浏览器为提升 JavaScript 解析速度，大量使用了 JIT 技术。

6.2 JIT 分类

根据 JIT 编译热点代码的粒度不同，JIT 可以分为两类，分别是 Method-based JIT 和 Trace-based JIT。下面就两类 JIT 分别介绍其特点。

6.2.1 Method-based JIT

所谓 Method-based JIT，是说筛选的热点代码的粒度是方法级的。Method-based 中的 Method 指 Java 代码中的方法或函数。Method-based JIT 通常在服务器端使用。在解释器执行过程中，解释器探测发现热方法，然后编译并优化被探测到的热方法。

由于 Method-based JIT 编译的是整个方法，除去函数参数和返回值外，函数内部和函数外没有数据依赖。这使得 JIT 在编译过程中更容易、更彻底地优化，编译后的 Native Code 的执行效率也就更高。但是 Method-based JIT 在编译和优化时也占用了更多的内存，且预热的速度相对来说比较慢，程序的高速运转需要前期较高的投入。

当前 Oracle 公司的 HotSpot 技术即是用了 Method-based JIT。在其中每个方法对应一个计数器，当其大于阈值(C1 为 1500，C2 为 10 000)时，表示该方法已经够“热”，并触发 HotSpot JIT 开始工作。

点拨 HotSpot VM 的编译器有 Client Compiler 和 Server Compiler，简称为 C1 和 C2 编译器。C1 编译器的优化比较快，C2 编译器能产生更好的代码，但是优化的速度就比较慢。

6.2.2 Trace-based JIT

同样地，Trace-based JIT 筛选的热点代码是以 Trace 为粒度。Trace-based 中的 Trace 指的是一段代码序列。解释器探测发现“热”执行路径，编译并优化这一小段“热”的小块程序。

相比于 Method-based JIT，其有优化粒度较细、额外内存开销小、编译热代码速度快、与解释器之间的过渡平滑等优点。和 Method-based JIT 不同，前期预热快，可以非常快地体现出性能提升。这也是为什么其适合嵌入式或手持设备等资源比较紧张、功耗要求高的平台。但同时，因其细粒度编译和优化，无法有顶峰的执行速度；频繁地在由解释器执行和执行 Native Code 之间转换需要更多的状态同步。

而 Dalvik VM 使用的平台基于电池；相比于 PC，内存资源和 CPU 资源略显不足；需要考虑到自身的安全性；在手持设备上需要快速地达到加速效果以及平滑的过渡于解释器和 Native Code 之间。因而，最终其采用的是 Trace-based JIT。Google 在 2010 年 I/O 会议上给出了一份统计数据，如图 6.1 所示。

从图中可以看到，虽然 Method-based JIT 有更好的优化窗口，但是 Trace-based JIT 需要的内存更少，编译的速度更快。

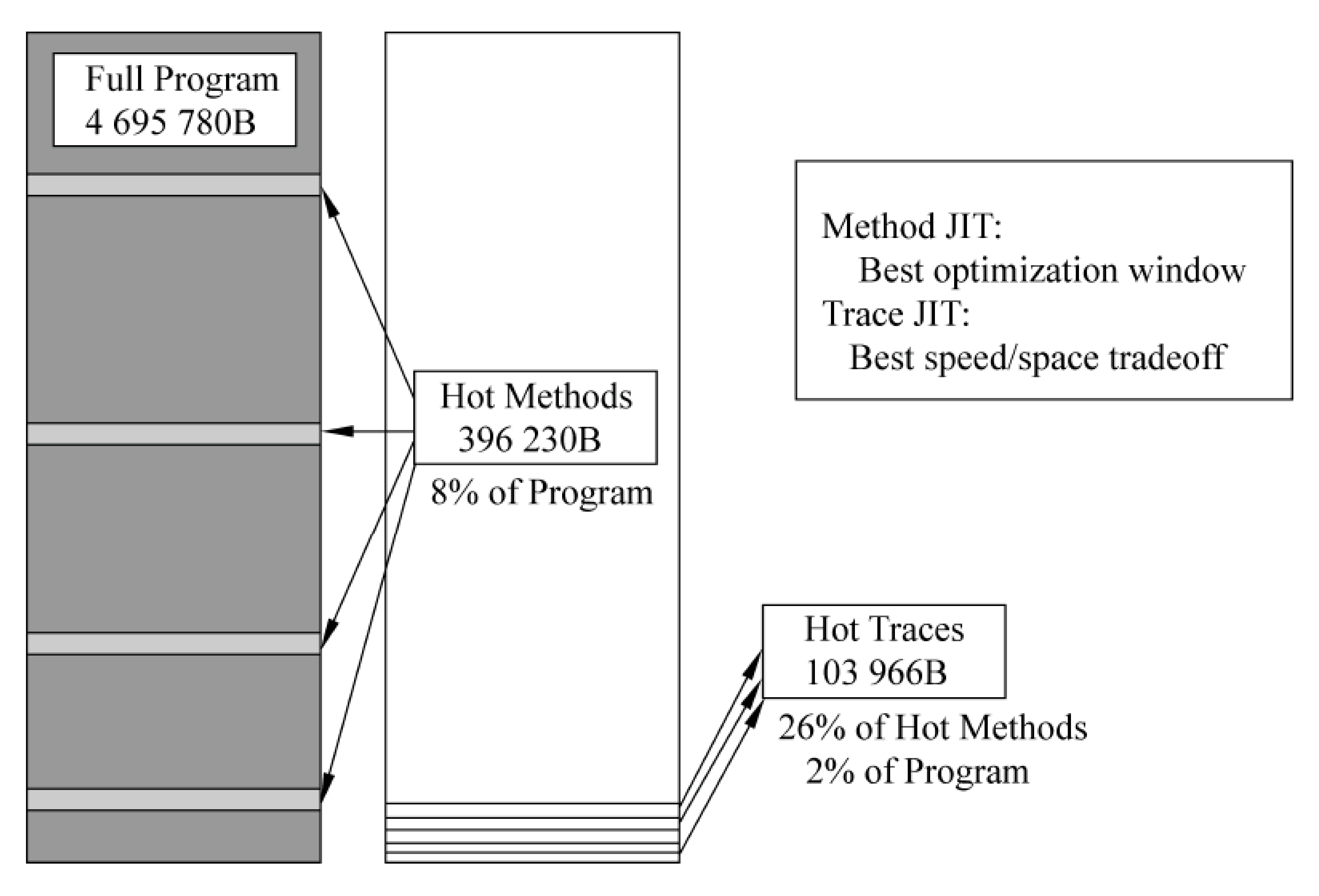


图 6.1　热方法和热路径的比例

6.3　整体框架分析

Dalvik 虚拟机从 Android 2.2 开始支持 JIT，以一个组件的形式增加了 JIT 编译模块，并可在编译时选择是否开启。Android 应用程序的执行性能由此提高了 2～5 倍。因为 Dalvik JIT 执行环境的特殊性(移动平台、手持设备、受电池限制)，需要平衡以下几个限制：①相比原来的解释器，尽可能地减少内存开销；②和 Dalvik 基于沙盒的安全模型兼容；③尽快地提高执行性能；④平滑地在解释器和编译后的代码之间过渡。考虑到以上因素，Dalvik JIT 采用的是 Trace 粒度的编译方式。

Dalvik VM 中 JIT 编译器只和解释器之间有比较紧密的耦合，且两者是属于不同的线程。解释器在解释时筛选出频繁执行的代码段，构造路径，并交由 JIT 编译器编译成本地代码；编译后的代码缓存到内存中，下次执行到相同的字节码时，可以直接执行编译好的本地代码。如图 6.2 所示，Dalvik JIT 编译器包含基本块构造、SSA 转换、优化、MIR 转 LIR 以及 LIR 转换为 Native Code 这几个部分。其中 Dalvik JIT 实现了包括寄存器分配、冗余内存操作指令消除、冗余有效性检查消除等优化，同时，特别优化了循环。其主要的调用入口为 dalvik/compiler/Frontend.cpp 中的 dvmCompilerTrace 函数。

点拨　Dalvik 内部包含许多线程，同时为保证启动效率，所有线程并不是同时建立的，具体请阅读 dalvik/vm/Init.cpp 中的初始化过程。

Dalvik JIT 在 Zygote 启动后，通过 dvmCompilerStartup 函数建立 JIT 线程并初始化 JIT。VM 为 Dalvik JIT 维护一个 gDvmJit 结构体，其中包含保存 JIT 入口地址的 Hash 表、保存二进制代码的入口地址等重要的信息。Dalvik VM 关闭时，该结构体将被释放。结构体定义如代码清单 6.1 所示。

代码清单 6.1　dalvik/vm/Globals.h：DvmJitGlobals：DvmJitGlobals

```
struct DvmJitGlobals {
    struct JitEntry * pJitEntryTable;              //入口表，每个 Trace 对应一个入口
```

```
    unsigned char * pProfTable;                //阈值计数器
    unsigned int jitTableSize;                 //可以使用的入口表大小
    unsigned int jitTableEntriesUsed;          //入口表使用情况
    void * codeCache;                          //编译后的代码存放地址
    unsigned int codeCacheSize;                //code cache 总大小
    unsigned int codeCacheByteUsed;            //已经使用的 code cache 大小
    …
}
```

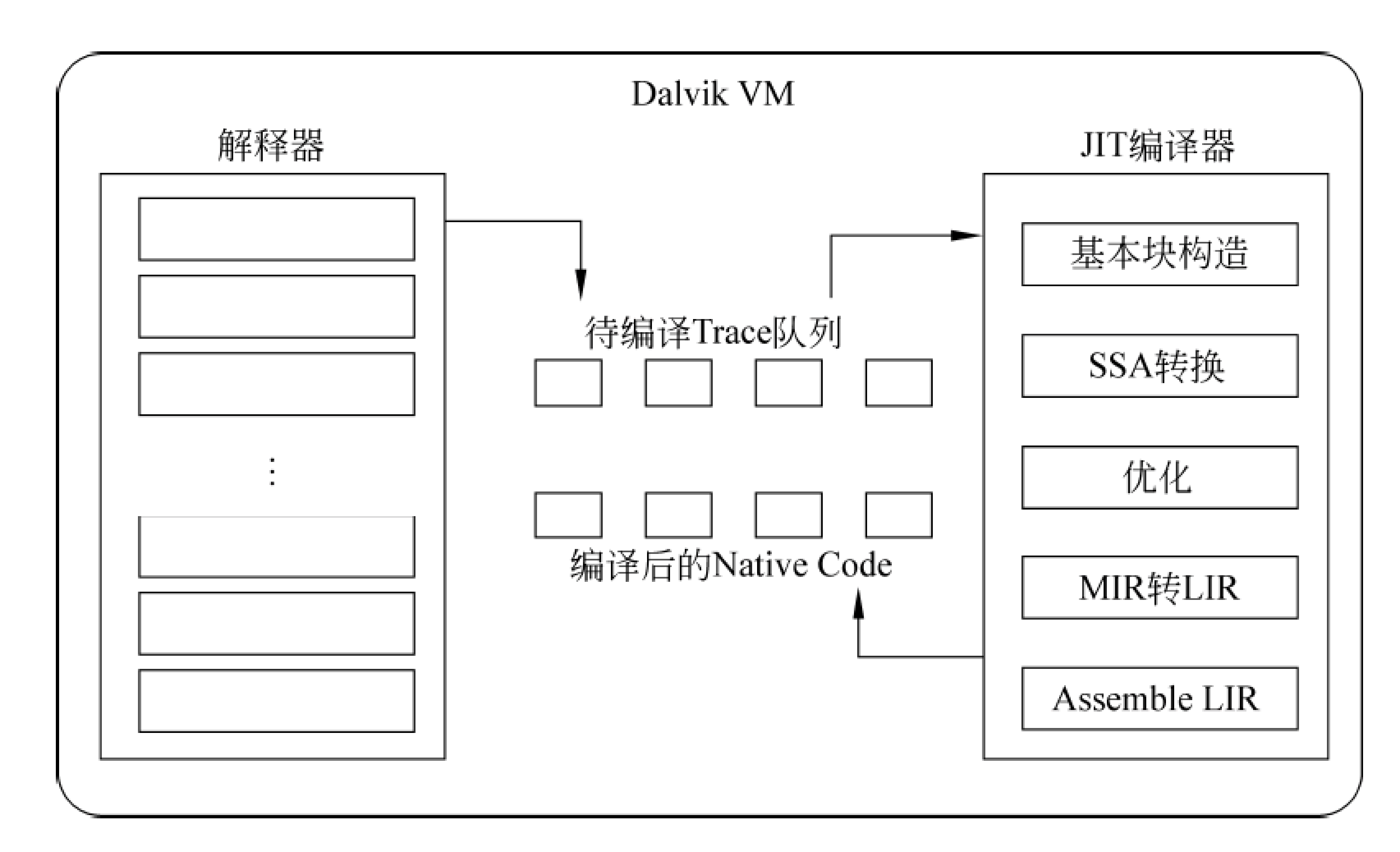


图 6.2 Dalvik VM 中解释器和 JIT 编译器间关系

工作环境设置好后，其工作框图如图 6.3 所示。

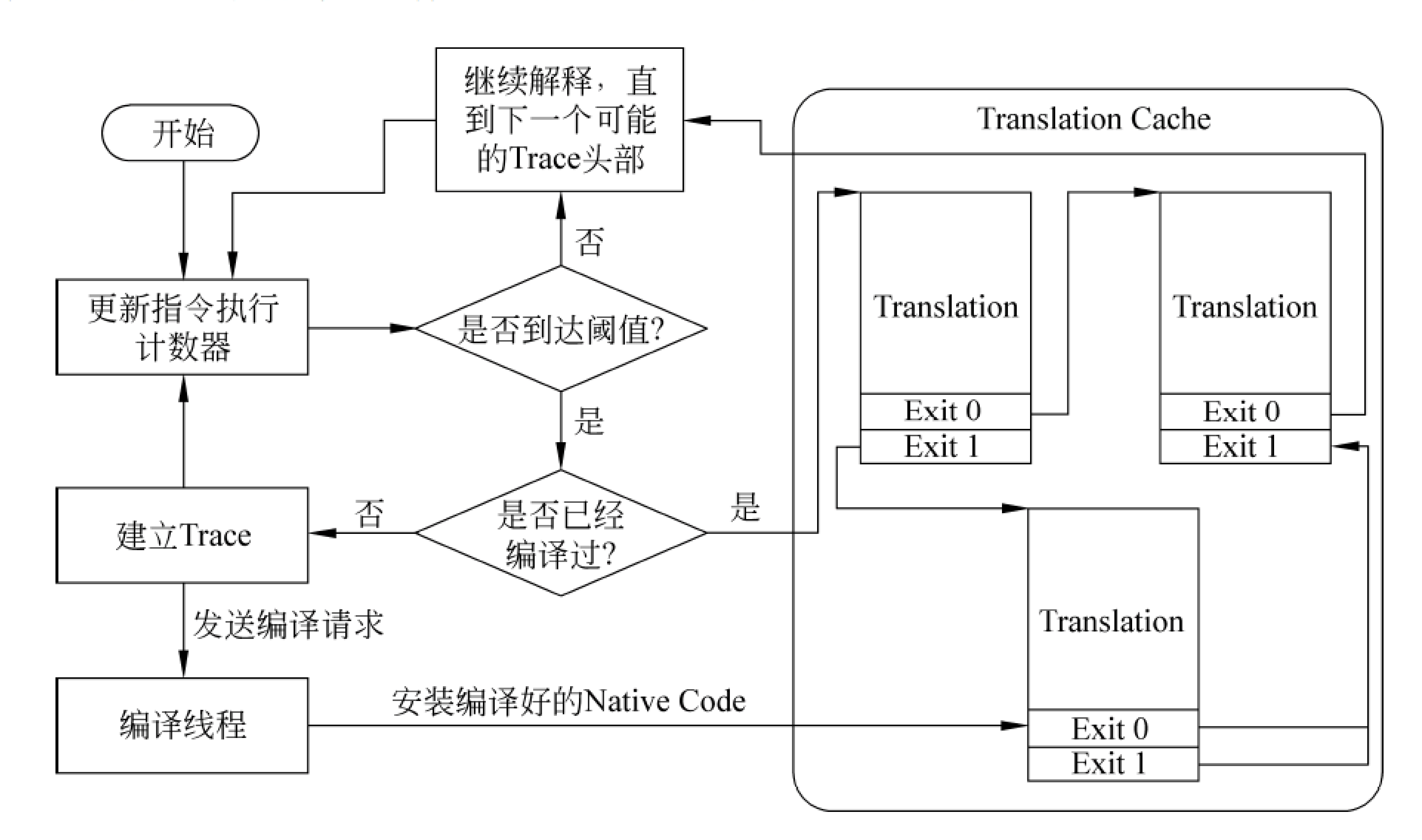


图 6.3 Dalvik JIT 工作框图

首先，先简略地解释下这个工作框图。在开始处，解释器正在解释执行字节码。当其识别到某些特定的指令时，更新这些指令对应的计数器；并调用 dvmCheckJit 函数检测这些指令的执行次数是否达到了阈值；达到阈值后，首先需要验证这段代码是否已被编译为二进制代码，对于没有被编译的 Trace，发送编译请求并构造 Trace。在此过程中，解释器执行的每

条字节码都需要指令检查,如果符合要求,就加入 Trace 中,直到遇到终止条件。路径构造结束后,将筛选出的 Trace 加入待编译队列。到这里为止,解释器也就结束了,并且开始正常解释字节码,不再进行指令检查。

接下来,让我们一窥这里涉及的一些设计和代码实现上的细节。

首先读者可能疑惑的是:什么是特定的指令?在构造 Trace 时,Dalvik 解释器识别的所谓特定指令如表 6.1 所示。从中可以看出这些指令都是跳转指令,Dalvik JIT 假定跳转指令之后的代码是热代码的可能性比较大。也就是这些指令之后执行的指令是“热代码”的可能性比较大。

表 6.1　识别指令

指　令	指 令 类 型
OP_GOTO OP_GOTO_16 OP_GOTO_32	GOTO 指令
OP_PACKED_SWITCH OP_SPARSE_SWITCH	SWIFCH 指令
OP_IF_EQ OP_IF_NE OP_IF_LT OP_IF_GE OP_IF_GT OP_IF_LE OP_IF_EQZ OP_IF_NEZ OP_IF_LTZ OP_IF_GEZ OP_IF_GTZ OP_IF_LEZ	IF 指令

那又怎么知道当前执行的指令是特殊的呢?在 Dalvik JIT 中,这些指令的识别并不需要判断,而是通过宏 WITH_JIT 控制,强制集成在解释器相应的汇编代码中,每次执行之时都需要执行判断是否到达阈值的程序。比如对于 OP_GOTO 指令,如代码清单 6.2 所示。

代码清单 6.2　dalvik/vm/mter p/InterpAsm-armv7-neon. S

```
.L_OP_GOTO: /* 0x28 */
/* File: armv5te/OP_GOTO.S */
    …
#if defined(WITH_JIT)
    ldr     r0,[rSELF,#offThread_pJitProfTable]
    bmi     common_testUpdateProfile     @ (r0) check for trace hotness
#endif
    …
```

接下来的问题是：如何知道这些指令执行的次数？Dalvik VM 维护了一张哈希表，以这些指令的 PC 值为 Hash 索引，存储了对应的执行次数。需要注意的是，阈值在 Hash 表中并不是从 0 开始，而是以阈值开始，逐渐递减。当 Hash 表中存储的计数器递减到 0 时，表示已经达到构建 Trace 的要求。以 armv7-a-neon 平台为例，Hash 表存储于 gDvmJit 中，名为 pJitProftable。将 PC 值加载到 r3 寄存器后，移位计算并加载至 r1 寄存器，以 armv7-a-neon 平台为例，这部分代码在 common_updateProfile 标签处，如代码清单 6.3 所示。

代码清单 6.3 dalvik/vm/mter p/InterpAsm-armv7-neon. S

```
/** Hash 阈值和 pc 值有关 * /
    eor     r3,rPC,rPC,lsr #12 @ cheap,but fast hash function
    lsl     r3,r3,# (32 - JIT_PROF_SIZE_LOG_2)          @ shift out excess bits
/* 取得阈值 * /
    ldrb    r1,[r0,r3,lsr # (32 - JIT_PROF_SIZE_LOG_2)] @ get counter
    GET_INST_OPCODE(ip)
/** 更新阈值,递减 * /
    subs    r1,r1,#1                                    @ decrement counter
/** 回存 * /
    strb    r1,[r0,r3,lsr # (32 - JIT_PROF_SIZE_LOG_2)] @ and store it
/**判断是否达到阈值,没达到则回到解释器,继续解释执行 * /
    GOTO_OPCODE_IFNE(ip)                         @ if not threshold,fallthrough otherwise * /
```

其中在 ARM 汇编中涉及的一个变量是 offThread_pJitProfTable，在 C 中对应的变量是 pJitProfTable。这是个数组名称，也就是上述说过的存储 pc 对应的 threshold 值。

不同的平台具有不一样的阈值，主流几个平台的阈值如表 6.2 所示。

指令到达阈值后，解释器需要判断 Trace 是否被编译。验证代码段是否被编译过的过程是，查找 Hash 表，如果 Trace 已经被编译，则地址不为空，直接跳转执行，否则解释器继续执行，编译线程继续工作。如果没有编译，将置位解释器中相应标志位，解释器发出构造 Trace 请求。这个过程如图 6.4 所示。

表 6.2 识别指令

平　台	阈　值
ARM v5te	200
ARM v5te-vfp	200
ARM v7-a	40
ARM v7-a-neon	40
x86	200

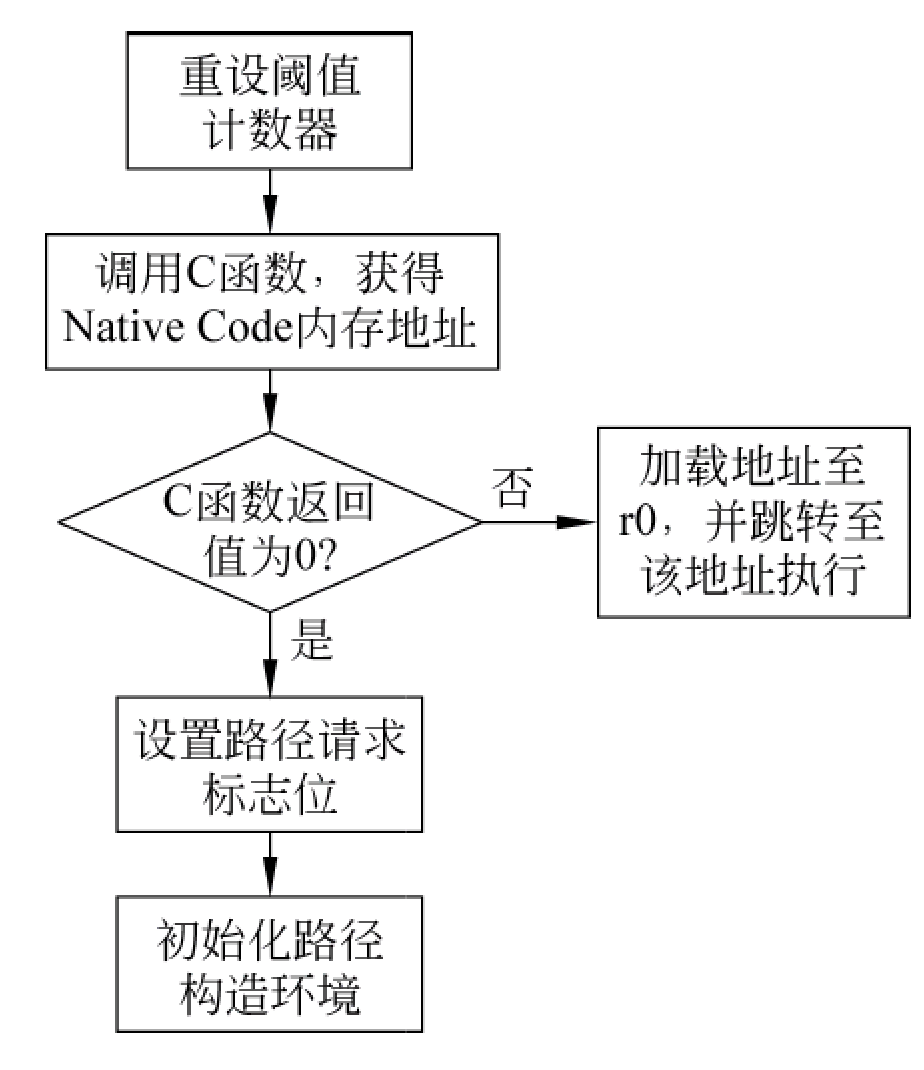


图 6.4 指令到达阈值后的处理

其中涉及的主要代码如代码清单 6.4 所示。

代码清单 6.4 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
/* 达到阈值,将计数器重新置位 * /
    ldr     r1,[rSELF,#offThread_jitThreshold]
    strb    r1,[r0,r3,lsr #(32 - JIT_PROF_SIZE_LOG_2)]      @ reset counter
    EXPORT_PC()
/* 调用函数,根据 pc 值,获取二进制代码所在内存地址,r0、r1 存储函数参数 * /
    mov     r0,rPC
    mov     r1,rSELF
    bl      dvmJitGetTraceAddrThread                        @ (pc,self)
/* 存储内存地址,inJitCodeCache 是线程中的一个指针类型(void * ),如果其为 NULL,则现在是解释执
行 * /
    str     r0,[rSELF,#offThread_inJitCodeCache]   @ set the inJitCodeCache flag
    mov     r1,rPC                                 @ arg1 of translation may need this
    mov     lr,#0                                             @ in case target is HANDLER_INTERPRET
/* 判断内存地址是否是 0 **/
cmp         r0,#0
    moveq   r2,#kJitTSelectRequest                 @如果不是 0,则发送构造路径请求
    beq     common_selectTraceJIT                  @跳转执行路径选择
@跳转到路径选择后,主要需要初始化一些变量,代码如下
common_selectTrace:
    …
/* 这里需要调用 dvmJitCheckTraceRequest 函数(C语言编写)进行环境的初始化,
 * 初始化之后,就直接进入了单步构造路径模式了。因此在初始化之前,需要保存
 * 当前的一些环境(主要是 pc 和 fp 指针)
 * /
    EXPORT_PC()
    SAVE_PC_FP_TO_SELF()                           @ copy of pc/fp to Thread
    bl                                             dvmJitCheckTraceRequest
    …
```

初始化之后,解释器继续解释指令,不同的是,在解释每条指令之前,都会调用 dvmCheckBefore 进行指令检查,判断是否要加入编译队列中。比如对于 MOV 指令,其执行代码如代码清单 6.5 所示。

代码清单 6.5 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S

```
.L_ALT_OP_MOVE: /* 0x01 * /
    …
    mov     r0,rPC              @ arg0
    mov     r1,rFP              @ arg1
    mov     r2,rSELF            @ arg2
    b       dvmCheckBefore      @ (dPC,dFP,self) tail call
```

dvmCheckBefore 函数的函数体非常大,除了处理 JIT 筛选路径外,还需要处理调试等其他信息。该函数当遇到下述几种情况时,Trace 建立就结束了。

(1) 如果 Trace 中的指令数不为 0,且当前指令是 SWITCH 指令,则结束该 Trace;
(2) 如果不是 GOTO 指令,且是 Branch、Switch、Return、Invoke 指令,则结束该 Trace;
(3) 如果抛出异常(THROW)指令或者是个自身循环指令,则结束该 Trace;
(4) Trace 中的指令数超出限制,结束(每个 Trace100 条指令);
(5) 如果是 Return 指令,返回非空,结束。

至此,Trace 的筛选也就结束了。

编译线程所做的工作是不断扫描队列,查看编译队列中是否有未编译 Trace。如果队列非空,取出队列中首条 Trace 并调用 dvmCompilerDoWork 函数编译之。编译生成的 Native Code 需要安装在 Translation Cache 中。这部分工作由 compilerThreadStart 函数完成。代码如代码清单 6.6 所示。

代码清单 6.6 dalvik/vm/compiler/Compiler.cpp:compilerThreadStart()

```
while (!gDvmJit.haltCompilerThread) {
    /**首先需要判断待编译队列中是否有未编译 Trace* /
    if (workQueueLength()==0) {
        int cc;
        cc=pthread_cond_signal(&gDvmJit.compilerQueueEmpty);
        assert(cc==0);
        /**如果没有就等待 * /
        pthread_cond_wait(&gDvmJit.compilerQueueActivity,
                          &gDvmJit.compilerLock);
        continue;
    } else {
        do {
            /**从队列中取出元素 * /
            CompilerWorkOrder work=workDequeue();
            dvmUnlockMutex(&gDvmJit.compilerLock);

            if (gDvmJit.haltCompilerThread) {
                LOGD("Compiler shutdown in progress - discarding request");
            } else if (!gDvmJit.codeCacheFull) {
                jmp_buf jmpBuf;
                work.bailPtr=&jmpBuf;
                bool aborted=setjmp(jmpBuf);
                if (!aborted) {
                    /**编译 Trace* /
                    bool codeCompiled=dvmCompilerDoWork(&work);
                    /**设置跳转地址 * /
                    dvmLockMutex(&gDvmJit.compilerLock);
                    if ((work.result.cacheVersion==
                        gDvmJit.cacheVersion) &&
                        codeCompiled &&
                        !work.result.discardResult &&
```

```
                    work.result.codeAddress) {
                    dvmJitSetCodeAddr(work.pc,work.result.codeAddress,
                                      work.result.instructionSet,
                                      false,/* not method entry */
                                      work.result.profileCodeSize);
                }
                dvmUnlockMutex(&gDvmJit.compilerLock);
            }
            /**重置分配的内存 */
            dvmCompilerArenaReset();
        }
        free(work.info);
        dvmLockMutex(&gDvmJit.compilerLock);
    } while (workQueueLength() != 0);
  }
}
```

Dalvik VM 执行 Native Code 时，解释器设置标志位，解释器线程将挂起。最后，二进制代码执行结束后，跳回解释器继续解释。

Translation Cache 是内存中开辟出专门用来存放编译 Trace 后得到的机器码，对应于 gDvmJit 中的 codeCache。Dalvik VM 同时维护了另一张以 PC 值作索引的 Hash 表 pJitEntryTable，存储了对应 Trace 的编译后的信息（包括在内存中的存储位置）。

从 Native Code 跳回解释器时，需要恢复解释器环境，包括：①PC；②FP；③线程基指针；④解释器标志位。并且包含 6 种跳回模式，如表 6.3 所示。

表 6.3　Native Code 跳回解释器模式

名　称	特　点
dvmJitToInterpNormal	如果 Dalvik 中的 PC 指针有对应的已编译的 Trace，则将该 Trace 和原 Trace 连接并跳到该 Trace 执行，否则，跳转到该 pc，解释执行
dvmJitToInterpNoChain	和 Normal 类似，只是没有连接
dvmJitToInterpPunt	跳转到 Fast 解释器解释指令，直到返回编译代码，处理异常所用
dvmJitToInterpSingleStep	使用 Portable 解释器解释下一条指令，解释完后回到 Native Code。用于处理单步调试
dvmJitToInterpTraceSelect	和 Normal 类似，不同在于 Dalvik 中 PC 指针对应的路径不存在，则申请构造路径
dvmJitToInterpBackwardBranch	为 SELF_VARIFICATION 特制，PC 在自身 Trace 中

6.4　前端功能及原理分析

Dalvik JIT 前端主要包筛选 Trace、构造基本块、确定控制流关系以及数据流分析及优化这 4 个部分。前端的输入是 Dalvik 字节码，输出是 SSA 形式的由基本块组成的控制流图。流程如图 6.5 所示。本节就以这 4 个部分来介绍 Dalvik JIT 前端。Dalvik JIT 对循环做了专门的优化，在后两个部分和不包含循环的处理有所不同，比如虽然都有 SSA 的转换，

但是不包含循环的 Trace 缺少一些优化等。对于不包含循环的 Trace，编译入口为 dalvik/compiler/Frontend.cpp 中的 dvmCompilerTrace，而针对循环，入口则为 commonLoop。本节主要以一个循环为例贯穿 Dalvik JIT 前端过程。

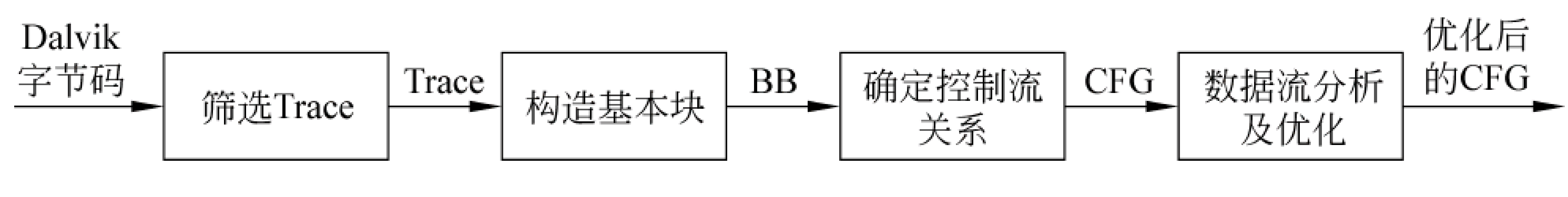


图 6.5 Dalvik JIT 前端流程

循环代码转换为字节码后如代码清单 6.7 所示。

代码清单 6.7 循环代码转换为字节码后的代码

```
0000: const/4 v0,# int 0           // # 0
0001: move v1,v0
0002: const/16 v2,# int 1000       // # 3e8
0004: if-ge v1,v2,000a             // + 0006
0006: add-int/2addr v0,v1
0007: add-int/lit8 v1,v1,# int 1   // # 01
0009: goto 0002                    // - 0007
000a: return-void
```

6.4.1 构造基本块

编译线程在 Dalvik VM 启动后启动，之后会不断检测到编译队列中是否有待编译的 Trace。检测到编译队列有待编译的 Trace 后，开始编译的第一步，构造基本块(Basic Block，BB)。基本块是一个只能从它的第一条指令进入，并从最后一条指令离开的最长的指令序列。因此，其第一条指令只能是这三种情况：①例程的入口点；②分支的目标；③直接紧跟在分支指令或跳转指令之后的指令。

构造基本块主要分为两部分，第一部分为 Dalvik JIT 在构造路径时候按照基本块原则将 Trace 划分为多个 TraceRun。dvmCheckJit 如代码清单 6.8 所示。

代码清单 6.8 dalvik/vm/interp/Jit.cpp:dvmCheckJit()

```
/**当前执行的指令已经不是顺序执行了 * /
if (lastPC != self->currRunHead + self->currRunLen) {
    int currTraceRun;
    /* We need to start a new trace run * /
    currTraceRun=++self->currTraceRun;
    self->currRunLen=0;
    self->currRunHead=(u2*)lastPC;
    self->trace[currTraceRun].info.frag.startOffset=offset;
    self->trace[currTraceRun].info.frag.numInsts=0;
    self->trace[currTraceRun].info.frag.runEnd=false;
    self->trace[currTraceRun].info.frag.hint=kJitHintNone;
    self->trace[currTraceRun].isCode=true;
}
```

```
self->trace[self->currTraceRun].info.frag.numInsts++;
self->totalTraceLen++;
self->currRunLen += len;
```

剩余的部分则由编译线程完成，其主要工作是：

（1）遍历 Trace 中指令；

（2）转换 Dalvik 字节码为 MIR；

（3）将 MIR 添加到基本块中，并判断是否是跳转指令，如果是，新建基本块，并标记为当前正使用基本块，否则继续执行；

（4）在跳转指令后添加 Chaining Cell。此时该类型基本块中没有任何代码。

这部分的代码在 dvmCompilerTrace 函数中，如代码清单 6.9 所示。

代码清单 6.9 dalvik/vm/compiler/Frontend.cpp:dvmCompilerTrace()

```
while (1) {
    MIR * insn;
    int width;
    insn= (MIR * )dvmCompilerNew(sizeof(MIR),true);
    insn->offset= curOffset;
    /**解析并分解指令,设置 MIR中相应元素 * /
    width= parseInsn(codePtr,&insn->dalvikInsn,cUnit.printMe);

    assert(width);
    insn->width= width;
    traceSize+=width;
    /**将 MIR加入到基本块中,curBB为当前正在使用基本块 * /
    dvmCompilerAppendMIR(curBB,insn);
    cUnit.numInsts++;
    /**获取指令类型 * /
    int flags= dexGetFlagsFromOpcode(insn->dalvikInsn.opcode);
    /**指令是调用指令时的处理 * /
    if (flags & kInstrInvoke) {
        const Method * calleeMethod= (const Method * )
            currRun[JIT_TRACE_CUR_METHOD].info.meta;
        assert(numInsts==1);
        CallsiteInfo * callsiteInfo=
            (CallsiteInfo * )dvmCompilerNew(sizeof(CallsiteInfo),true);
        callsiteInfo->classDescriptor= (const char * )
            currRun[JIT_TRACE_CLASS_DESC].info.meta;
        callsiteInfo->classLoader= (Object * )
            currRun[JIT_TRACE_CLASS_LOADER].info.meta;
        callsiteInfo->method= calleeMethod;
        insn->meta.callsiteInfo= callsiteInfo;
    }

    /**指令数超过临界值了 * /
```

```
        if (cUnit.numInsts >= numMaxInsts) {
            break;
        }
        /**在当前 TraceRun 中,已经是最后一条指令 * /
        if (--numInsts==0) {
            if (currRun->info.frag.runEnd) {
                break;
            } else {
                /** 寻找包含代码的 TraceRun * /
                do {
                    currRun++;
                } while (!currRun->isCode);

                /* Dummy end-of-run marker seen * /
                if (currRun->info.frag.numInsts==0) {
                    break;
                }
                /**新建一个基本块 * /
                curBB=dvmCompilerNewBB(kDalvikByteCode,numBlocks++);
                dvmInsertGrowableList(blockList,(intptr_t) curBB);
                curOffset=currRun->info.frag.startOffset;
                numInsts=currRun->info.frag.numInsts;
                curBB->startOffset=curOffset;
                codePtr=dexCode->insns+curOffset;
            }
        } else {
            curOffset+=width;
            codePtr+=width;
        }
    }
```

可以看到,编译线程主要的工作就是调用 parseInsn 函数将字节码转换为 MIR。MIR 全称为中级中间表示。MIR 是拆解了的 Dalvik 字节码,在形式上接近 Dalvik 字节码,为便于后期的方便操作。其包含 Dalvik 字节码,并包括字节码的一些其他属性,比如字节码长度(width)。其结构体定义如代码清单 6.10 所示。

代码清单 6.10 dalvik/vm/compiler/CompilerIR.h:MIR

```
typedef struct MIR {
    DecodedInstruction dalvikInsn;      //操作码
    unsigned int width;                 //指令长度
    unsigned int offset;                //指令位置
    struct MIR * prev;                  //前一条 MIR 指令
    struct MIR * next;                  //后一条 MIR 指令
    ...
} MIR;
```

除去 Trace 本身划分的基本块之外，每一个 Trace 都还对应了一些额外的基本块，有如下几类：Entry Block（Trace 的入口）、Exit Block（Trace 的出口）、Backward Branch Block、PC Reconstruction Block、Exception Handling Block。Backward Branch Block 是因为循环新加的。

点拨 其实 MIR 并没有多少特殊，只是字节码分解后的产物。其实和普通指令拆解出操作码、操作数、指令长等信息在原理上是一样的。字节码、MIR 以及后文所说的 LIR 都是中间代码。

当 Trace 被确定可能包含循环后，Dalvik JIT 将会有专门的处理和优化，基本块的构造过程和前述略有不同。循环的基本块是递归构造的，如果 Trace 是嵌套循环的一部分，将会扫描出整个嵌套循环，还包括不在预先构建的 Trace 中的内容。同时，针对循环划分出的基本块其顺序安排也略有不同。Exit Block 是安排在 Entry Block 之后，而不是在最后。这部分内容由 dalvik/vm/compiler/exhaustTrace 函数完成。

点拨 Dalvik JIT 单独对包含循环的 Trace 进行处理，入口为 dalvik/vm/compiler/Frontend.cpp/compileLoop 函数中。需要注意的是 dvmCompilerTrace 跳转入这个函数前，需要释放编译 Trace 时分配的资源。因为对于循环，前面构造处的 Trace 并不完整。

本节开头处的字节码将至少被识别出一条 Trace，该 Trace 包含循环。基本块构造后，将会有两个代码块，分别包括指令 0006、0007、0009 三条指令和 0002、0004 两条指令。

6.4.2 确定控制流关系

分解完 Trace 并构建好基本块后，需要扫描基本块，确定基本块之间的连接关系，也就是 Trace 的控制流关系。

在编译器中，控制流图 $G=(N,E)$ 定义如下：

（1）包含节点集 N；

（2）包含边集 $E \subseteq N \times N$。通常，将边写成 $a \rightarrow b$，而不是 $<a,b>$。如果边 $x \rightarrow y \in E$，表示 x 节点执行完后可以马上执行 y 节点；

（3）包含一个入口节点 Entry 和一个出口节点 Exit，其中 Entr $\in N$，Exit $\in N$。

和 Dalvik JIT 相对应，基本块即是这里的节点，基本块之间的联系则对应了边集。一般情况下，Dalvik JIT 需要扫描每一个基本块，并分析每一个基本块的最后一条指令。根据指令跳转地址，插入基本块或连接基本块。Dalvik JIT 中每一个基本块都有 taken 和 fallthrough 指针，指向要跳转的基本块。

如果基本块的跳转地址小于当前指令的地址，初步认为其是可以优化的循环，Dalvik JIT 会对其进行专门的处理。Android 4.0.4 版本中的 Dalvik JIT 在开始处理循环时，会销毁开始已经分配的内存（如基本块），并重置已经设置过的变量（如基本块个数）。确定包含循环的 Trace 的控制流图是和构造 Trace 基本块融合在一起的。在递归地构造基本块的同时，会连接各个基本块。

代码清单 6.11 dalvik/vm/compiler/Frontend.cpp:dvmCompileTrace#()

```
for (blockId=0; blockId<blockList->numUsed; blockId++) {
    curBB=(BasicBlock *) dvmGrowableListGetElement(blockList,blockId);
    MIR * lastInsn=curBB->lastMIRInsn;
```

```
        /**略过空块 * /
        if (lastInsn= = NULL) {
            continue;
    }
    /** 计算 fallthrough 和 target 地址,target 地址为最后一条指令跳转的地址,
    * 相应的 fallthrough 为顺序存储指令的下一条指令 * /
        curOffset= lastInsn- > offset;
        unsigned int targetOffset= curOffset;
        unsigned int fallThroughOffset= curOffset + lastInsn- > width;
        bool isInvoke= false;
        const Method * callee= NULL;

        findBlockBoundary(desc- > method,curBB- > lastMIRInsn,curOffset,
                                &targetOffset,&isInvoke,&callee);

        /* Link the taken and fallthrough blocks * /
        BasicBlock * searchBB;

        int flags= dexGetFlagsFromOpcode(lastInsn- > dalvikInsn.opcode);
        /**记录 invoke 指令,后续处理有用 * /
        if (flags & kInstrInvoke) {
            cUnit.hasInvoke= true;
        }

    /** 如果跳转指令跳转的地址小于当前指令的地址,则可能包含循环,进行专门
    * 的处理,在这之前,需要注意的是需要释放资源 * /
        if (isInvoke= = false &&
            (flags & kInstrCanBranch) != 0 &&
            targetOffset< curOffset &&
            (optHints & JIT_OPT_NO_LOOP)= = 0) {
            dvmCompilerArenaReset();
            return compileLoop(&cUnit,startOffset,desc,numMaxInsts,
                                    info,bailPtr,optHints);
        }

        /* 遍历所有的基本块,比较 target 和 fallthrough 与基本块首条指令的偏移地址 * /
        size_t searchBlockId;
        for (searchBlockId= blockId+ 1; searchBlockId< blockList- > numUsed;
             searchBlockId+ + ) {
            searchBB= (BasicBlock * ) dvmGrowableListGetElement(blockList,
                                                              searchBlockId);
            if (targetOffset= = searchBB- > startOffset) {
                curBB- > taken= searchBB;
                dvmCompilerSetBit(searchBB- > predecessors,curBB- > id);
            }
```

```
            if (fallThroughOffset==searchBB->startOffset) {
                curBB->fallThrough=searchBB;
                dvmCompilerSetBit(searchBB->predecessors,curBB->id);
                /** 函数调用指令需要对齐处理 * /
                if (flags & kInstrInvoke) {
                    searchBB->isFallThroughFromInvoke=true;
                }
            }
        }
    /** 基本块中有些指令没有改变控制流,比如 fallthrough、target 跳转地址不
    * 在 Trace 中、switch 指令等,这些有不同的处理,篇幅原因,就省略部分代码 * /
        curBB->needFallThroughBranch=
            ((flags & (kInstrCanBranch | kInstrCanSwitch | kInstrCanReturn |
                    kInstrInvoke))==0);
        if (lastInsn->dalvikInsn.opcode==OP_PACKED_SWITCH ||
            lastInsn->dalvikInsn.opcode==OP_SPARSE_SWITCH) {
            …
        /* fallthrough 跳转的地址不在 Trace 中,此时需要跳回解释器 * /
        } else if (!isUnconditionalBranch(lastInsn) &&
                curBB->fallThrough==NULL) {
            …
        }
        /* target 跳转地址不在 Trace 中 * /
        if (curBB->taken==NULL &&
            (isGoto(lastInsn) || isInvoke ||
            (targetOffset !=UNKNOWN_TARGET && targetOffset !=curOffset))) {
        …
            } else {/**最后对那些非条件跳转的指令进行处理 * /
                …
            }
            if (newBB) {/**新建一个 BB* /
                curBB->taken=newBB;
                dvmCompilerSetBit(newBB->predecessors,curBB->id);
                dvmInsertGrowableList(blockList,(intptr_t) newBB);
            }
        }
    }
```

对于本节开头的例子,将得到如图 6.6 所示的内存安排位置图以及如图 6.7 所示的基本块控制流图。为描述简便,图 6.7 经过简化后变为图 6.8。

6.4.3　识别及筛选循环

编译器在优化处理方面经过几十年的发展,已经形成了一套理论,而在 Dalvik JIT 中的很多优化实现都是基于前人的理论基础。因此,首先不得不引入一些枯燥的理论知识,解释在这个过程中涉及的一些基本概念。

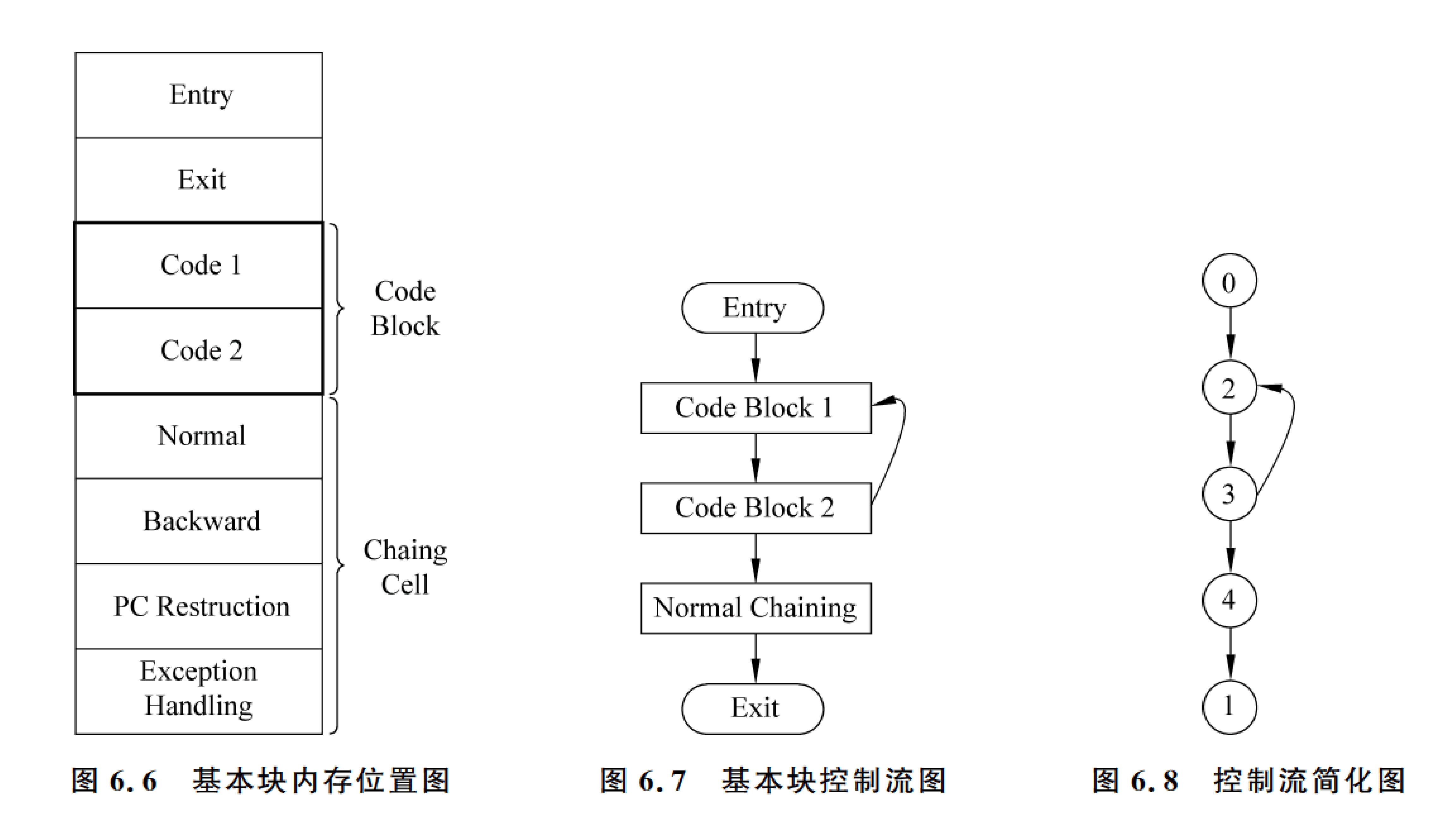


图 6.6　基本块内存位置图　　图 6.7　基本块控制流图　　图 6.8　控制流简化图

必经节点(Dominator)：如果从 Entry 节点到节点 i 的每一条可能的执行路径都包含 d 节点，则说节点 d 是节点 i 的必经节点，记作 d dom i。根据定义可以看到 dom 是自反的、传递的以及反对称的。记 i 的必经节点集合是 dom (i)。

严格必经节点(Strict Dominator)：如果 a dom b 且 $a \neq b$，那么称 a 是 b 的严格必经节点，记作 a sdom b。记 b 的严格必经节点是 sdom (b)。

直接必经节点(Immediate Dominator)：对于 $a \neq b$，a idom b 当且仅当 a dom b 且不存在一个 $c \neq a$ 并且 $c \neq b$ 的节点 c，使得 a dom c 且 c dom b。记 b 的直接必经节点为 idom (b)。直观上，直接必经节点是距离 b 最近的必经节点。同时，节点的直接必经节点是唯一的。

回边(Back edge)：如果边 $a \rightarrow b$ 是回边，则其头 b 是尾 a 的必经节点。

在大多数语言中，循环有多种描述方法。以 C 语言为例，其中可以由 for 循环、while 循环甚至可以用 goto 语句和标号来定义。但是从程序分析的角度看，在源码中循环的描述方式并不重要，重要的是这些循环是否可以被优化。自然循环(nation loop)就是一种可以优化的循环，其定义如下。

(1) 必须具有一个唯一的入口节点(Entry)，称为循环头。该节点支配了循环中的所有节点。

(2) 必然存在一条进入循环头的回边，否则就不存在循环。

已知一条回边 $a \rightarrow b$ 的自然循环由满足如下条件的节点集合和边集合组成：节点集合由节点 b 以及流图中那些不经过 b 可以到达 a 的所有节点组成，边集合由节点集合众连接节点的边组成。同时可知，节点 b 是自然循环的循环头，只有循环头是相同的，两个循环才可能一样。

Dalvik JIT 在筛选的循环首先必须满足自然循环这个条件，其次只筛选出一条回边的自然循环。正如前文所说，如果循环是嵌套的，在构造基本块和确定控制流关系时，将得到包含其中的所有循环。Dalvik JIT 利用必经节点(Dominator)来标识和筛选出 Trace 中的自然循环。这一部分的工作主要由函数 dvmCompilerBuildLoop 完成。

基于以上概念，Dalvik JIT 首先对整个控制流图进行深度优先排序(depth-first ordering)(函数 computeDFSOrder)。在深度优先排序中，首先访问一个节点，然后递归遍历该节点的右子节点，再递归遍历该节点的左子节点。以图 6.7 为例，深度优先排序的顺序为{0,2,3,4,1}。按照深度优先排序后的节点顺序计算节点间的 dominance 关系，每个节点的必经节点为：

$D(0)=\{0\}$

$D(2)=\{2\}\cup D(2)=\{0,2\}$

$D(3)=\{3\}\cup D(2)=\{0,3\}$

$D(4)=\{0,2,3,4\}$

$D(1)=\{0,1,2,3,4\}$

每个节点的直接必经节点为：

i dom (0)=Φ

i dom (2)={0}

i dom (3)={2}

i dom (4)={3}

i dom (1)={4}

由函数 compute|Dominators 确定节点间的 dominance 关系后，需要筛选出 Trace 中包含的循环。Dalvik JIT 默认循环头必须在入口节点指向的第一个节点，并且循环头至少需要前驱节点。之后，根据计算好的 dominance 关系，判断 Trace 中是否有回边。比如对于上述例子，边 3→2 就是该循环的回边。由于该例是个单重循环，经过筛选后，所得到的控制流图和图 6.7 是一样的。

涉及的代码在 dalvik/vm/compiler/SSATransformation.cpp 中。CompileLoop 函数调用 dvmCompilerBuildLoop 函数进行识别和优化循环。代码如代码清单 6.12 所示。

代码清单 6.12　dalvik/vm/compiler/SSATransformation.cpp: dvmCompilerBuildLoop()

```
bool dvmCompilerBuildLoop(CompilationUnit * cUnit)
{
    /* 计算深度优先排序 */
    computeDFSOrder(cUnit);
    /* 计算必经节点信息 */
    computeDominators(cUnit);

    if (gDvmJit.noFilterLoop==false) {
        if (dvmCompilerFilterLoopBlocks(cUnit)==false)
            return false;
        /* 重新计算深度优先排序 */
        computeDFSOrder(cUnit);
        /* 重新计算必经节点信息 */
        computeDominators(cUnit);
    }
    /* 接下来一部分是进行 SSA 转换 */
    dvmInitializeSSAConversion(cUnit);
```

```
    /* 计算 define 和 use 属性的寄存器 */
    computeDefBlockMatrix(cUnit);
    /* 插入 phi 节点 */
    insertPhiNodes(cUnit);
    /* SSA 转换 */
    dvmCompilerDataFlowAnalysisDispatcher(cUnit,dvmCompilerDoSSAConversion,
kPreOrderDFSTraversal,false /* isIterative */);
    /** 记录已经分配的寄存器 */
    cUnit->tempSSARegisterV= dvmCompilerAllocBitVector(cUnit->numSSARegs,
                                                       false);
    /* 插入 Phi 节点的参数 */
    dvmCompilerDataFlowAnalysisDispatcher(cUnit,insertPhiNodeOperands,
                                          kReachableNodes,
                                          false /* isIterative */);
    /**打印控制流图 */
    if (gDvmJit.receivedSIGUSR2 || gDvmJit.printMe) {
        dvmDumpCFG(cUnit,"/sdcard/cfg/");
    }
    return true;
}
```

以上代码中包括 6.4.4 节中介绍的 SSA 形式转换。不论 Trace 中是否包含循环,都要进行 SSA 转换,其中涉及的原理是一样的。

点拨 读者可能已经注意到,在上面的代码中会在/sdcard/cfg/目录下有一些动作。不错,这部分代码用来输出当前已经构造完成的控制流图,是笔者在阅读源码过程中加上的。在阅读源码的过程中,输出当前的执行状态,对阅读源码有十分大的帮助。其使用方法是在运行 dex 或 zip 文件时,加上-Xjitverbose 参数。完成后,将在/sdcard/cfg/目录下生成 dot 图。用 dot 工具转换后就可以直观看到控制流图了。注意,/sdcard/下没有 cfg 目录,需要手动建立。

6.4.4 SSA 形式转换

静态单赋值是一种高效的数据流分析技术。如果在某个过程内赋值的每一个变量作为赋值的目标只出现一次,则说这个过程是静态单赋值(Static Single Assignment,SSA)的形式。作为一种中间表示,SSA 形式能带来精确的使用-定义关系,从而简化了一些优化过程,包括常数传播、死代码删除、部分冗余消除以及寄存器分配优化等。现代编译器一般首先将给定表示转换为 SSA 形式,优化基于 SSA 形式操作,最后将 SSA 形式转换成原来的形式。

转换为 SSA 形式中间代码的标准操作是给每一个赋值的变量带上一个下标(Subscription)。为区分分支指令中对变量的多种赋值操作,在分支指令的汇合点插入 ϕ 函数,如果程序是线性的,其中没有循环控制指令,则不需要 ϕ 函数。ϕ 函数的参数个数为汇合点出变量的版本个数,每个参数(版本)和汇合点的特定控制流前驱对应。如图 6.9 所示经过转换后变成图 6.10。经过变换后,和变量 x 等同的有$\{x_1,x_2\}$,同理变量 w、y、z 也类似。但是在汇合点处,无法确定 y 到底使用哪一个值,因为不论是左侧分支还是右侧分支,都对变量 y 进行了定义。此时插入了 ϕ 函数,其参数是 y_1 和 y_2,返回为变量 y 的新版本 y_3。

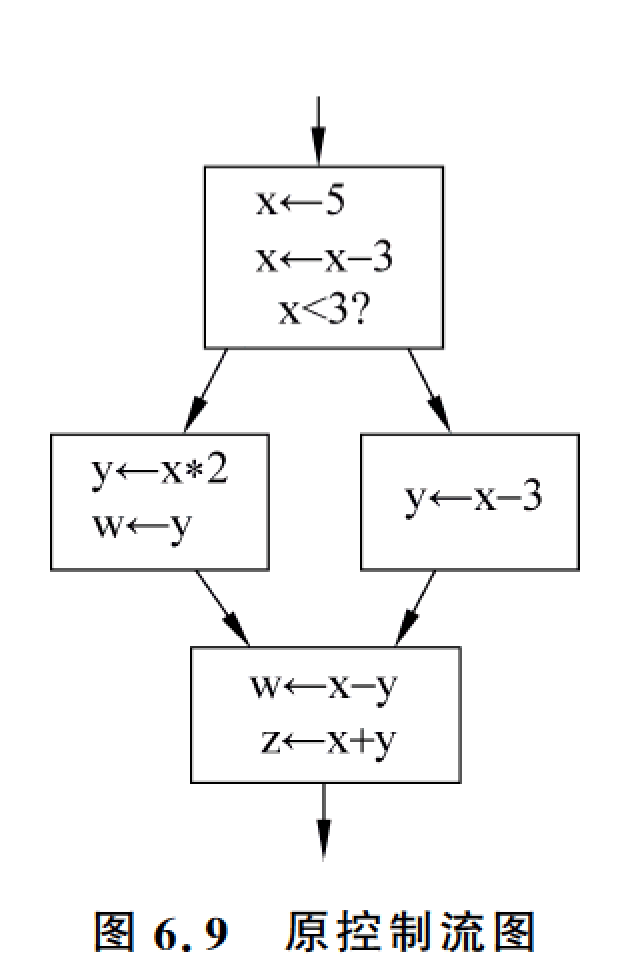


图 6.9　原控制流图

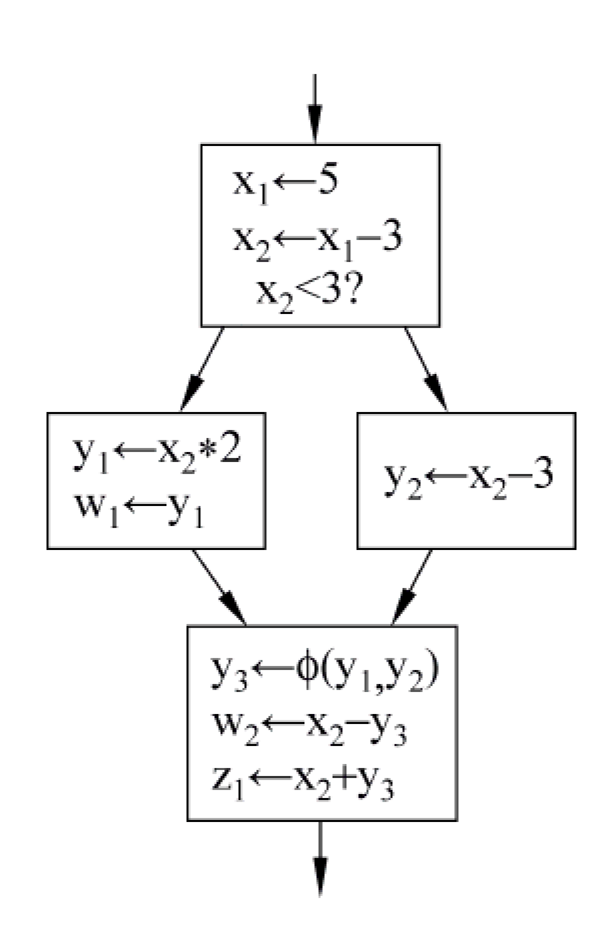


图 6.10　原控制流图

从普通的中间表示转换到 SSA 形式时需要以下两个步骤：①寻找汇合点，插入简单的 ϕ 函数（函数 insertPhiNodes）；②变量重命名（函数 dvmCompilerDoSSAConversion）。在 Dalvik JIT 中，这两部分的工作都在 dalvik/vm/compiler/SSATRansformation.cpp 中。寻找汇合点，插入 ϕ 函数一般借助支配边界来实现。在插入 ϕ 函数后，就开始进行变量重命名。

SSA 对寄存器的使用分为两种，分别是 Uses（使用）和 Define（定义）。Uses 表示只是对寄存器的使用，并没有更改寄存器中的值。Define 表示重新设置了寄存器中的值，下次再引用该寄存器会使用新的值。Dalvik 字节码对所有的字节码进行了分类，部分属性如表 6.4 所示。每一个字节码都有相应的属性。

表 6.4　数据流分析指令属性

属　性	含　　义	属　性	含　　义
kUA	Uses，vA	kDAWide	Define，vA wide
kUB	Uses，vB	kIsMove	Move 指令
kUC	Uses，vC	kSetsConst	Const 指令
kUAWide	Uses，vA wide（两个字节）	kFormat35c	35c 格式指令
kUBWide	Uses，vB wide	kFormat3rc	3rc 格式指令
kUCWide	Uses，vC wide	kPhi	Phi 节点
kDA	Define，vA		

比如指令 move vA，vB，其中使用 vA 寄存器，并定义了 vB 寄存器，因而指令包含 kUB 和 kDA 的属性。ϕ 函数定义了一个寄存器，因而其包含 kDA 和 kPhi 属性。但此时，因为不知道其他寄存器的 SSA 表示，ϕ 函数没有参数。

在转换成 SSA 形式时，需要将 Dalvik Reg（字节码中使用的寄存器）一一映射成 SSA Reg（SSA 形式使用的寄存器）；同时，在 SSA 形式转换回 Dalvik 字节码时，SSA 形式采用的寄存器表示需要映射回 Dalvik 寄存器。为实现这一功能，Dalvik JIT 设计了两个数据结构，分别是一个数组（Dalvik2SSAMap）和一个链表（SSA2DalvikMap）。

Dalvik VM 采用的 Dex 文件已经说明了整个 Dex 使用的寄存器个数。Dalvik2SSAMap 数

组的长度刚好是寄存器个数，存储的元素是 SSA 形式寄存器。Dalvik2SSAMap 数组元素为 32 位的整型，高 16 位表示下标，低 16 位为 SSA 形式寄存器号。同理，SSA2DalvikMap 中元素也是 32 位整型，高 16 位为下标，低 16 位为对应的 Dalvik 字节码中的寄存器号。

JIT 遍历代码块中所有 MIR，如果包含使用寄存器，则根据寄存器号从 Dalvik2SSAMap 中取出最近的 SSA 形式寄存器；如果包含定义寄存器，则根据寄存器号在 Dalvik2SSAMap 中设置新的 SSA 寄存器号，同时，在 SSA2DalvikMap 中添加新元素。

例如，指令 add-int/2addr v0，v0，v1 实现了将 v0＝v0＋v1 的功能，其 SSA 转换过程图如图 6.11 所示。首先转换有 USE 属性的寄存器，根据寄存器号在 Dalvik2SSAMap 中查找到当前正使用的 SSA 寄存器，此时 v0 和 v1 分别转换为 v5_0 和 v4_0。之后，再转换 Define 属性的寄存器 v0，根据寄存器号更改 Dalvik2SSAMap 中 SSA 寄存器 v5_0 为 v6_0；同时根据 SSA 寄存器号 6 在 SSA2DalvikMap 中添加对应的 Dalvik 寄存器 v0_2。Dalvik 寄存器 v0_2 表示使用 v0 寄存器，目前为第三个版本。其具体的实现过程在 dalvik/vm/compiler/Dataflow.cpp 中的 dvmCompilerDoSSAConversion 函数中。

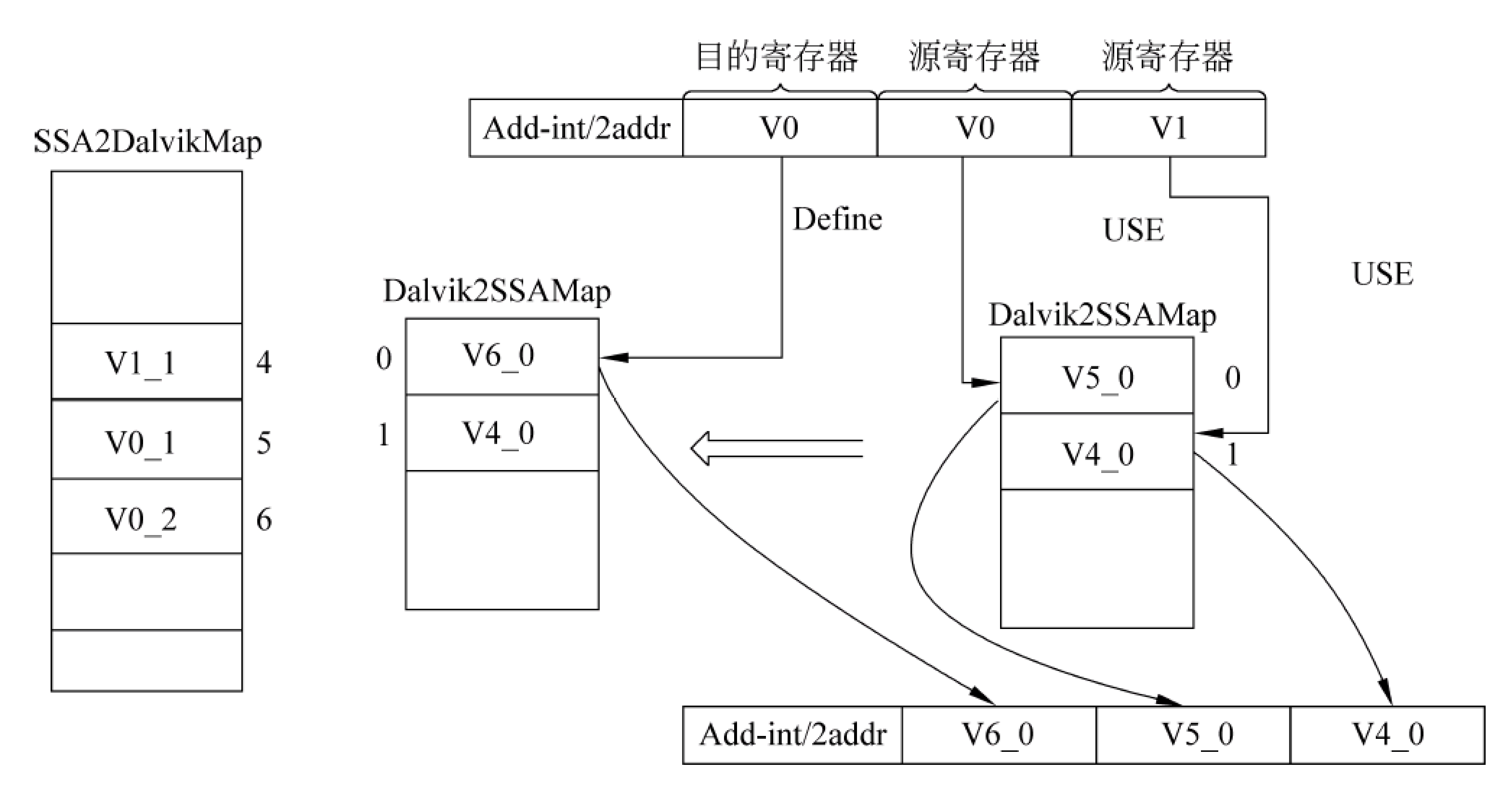


图 6.11　SSA 形式转换图

在 SSA 转换完成后，就可以进行基于 SSA 形式的数据流分析。Dalvik JIT 中包括常量传播、归纳变量、冗余边界检查消除等数据流分析及优化技术。由于 Dex 文件是由 Class 文件经过 dx 工具转换和优化过后生成的，JIT 所能做的工作并不是很多。

6.5　后端功能及原理分析

Dalvik JIT 后端主要包括寄存器分配、MIR 转换为 LIR、LIR 转换为机器码以及二进制 Native Code 的安装这 4 个部分。经过 Dalvik JIT 前端处理后，此时的输入是 SSA 形式的由基本块组成的控制流图。再经过寄存器分配以及进一步的优化转为机器码，最后安装到 Translation Cache 中，其流程图如图 6.12 所示。为简便叙述，本节将分为 MIR 转换为 LIR 和 LIR 转换为机器码两部分介绍。其中，MIR 转换为 LIR 包括寄存器分配；LIR 转换为机器码包括二进制代码的安装。

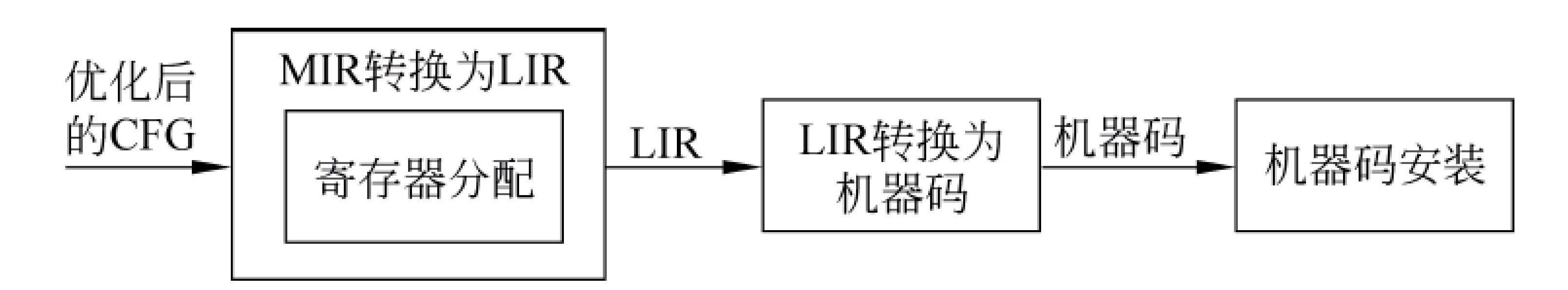


图 6.12　Dalvik JIT 后端流程

6.5.1　MIR 转换为 LIR

经过前端处理后，此时的输入是以 MIR 组成的 SSA 形式的控制流图(CFG)。输出是 LIR 链。所谓的 LIR，是和硬件平台相关的。由于篇幅限制，本章只是介绍针对 ARM 平台的后端过程。比如针对 ARM 平台的 LIR，其定义如代码清单 6.13 所示。

代码清单 6.13　dalvik/vm/compiler/codegen/arm/ArmLIR.h：ArmLIR

```
typedef struct ArmLIR {
    LIR generic;                    //通用 LIR
    ArmOpcode opcode;               //ARM指令操作码
    int operands[4];                // [0..3]= [dest,src1,src2,extra]
    …
} ArmLIR;
```

JIT 逐个基本块逐条扫描 MIR 指令，根据不同的指令类型进行相应的处理。每一个基本块对应了一个标签(Label)，标签名包含基本块的首条指令以保证标签的唯一性。

Dalvik JIT 为每一类的字节码都设计了转换函数，具体的函数实现在 dalvik/vm/compiler/codegen/arm/CodegenDriver.cpp 中。字节码根据第 1 卷第 2 章所述的字节码 ID 来分类，相同类别的指令有相同的转换函数。转换函数的命名规范是“handleFmt＋指令格式”，比如对格式为“11x”的指令，有函数如下：

```
handleFmt11x(CompilationUnit * cUnit,MIR * mir);
```

如果两类指令的处理过程类似，则在函数后再扩展上相应的格式。如对于格式为“35ms”和“3rms”的指令，其处理过程类似，则有：

```
handleFmt35ms_3rms(CompilationUnit * cUnit, MIR * mir, BasicBlock * bb, ArmLIR * labelList);
```

对于每一条指令，转换函数主要处理步骤如下。

(1) 获取源寄存器数据(如果有)并加载至 LIR 寄存器；

(2) 获取目的寄存器数据并加载至 LIR 寄存器；

(3) 根据 Dalvik 字节码指令操作码，生成对应的 LIR 指令；

(4) 运算结果存回目的寄存器对应的内存地址。

源寄存器或目的寄存器指的是 Dalvik 字节码中的寄存器。在加载 Dalvik 寄存器中的值至 LIR 寄存器的过程中，涉及 Dalvik JIT 转换后的 Native Code 和解释器之间的数据交互。在解释器中，某几个寄存器有特定的安排，如表 6.5 所示。

比如对于 MOVE_RESULT 指令，就有如下代码，其中 rlDest 是目的寄存器，rlSrc 是源寄存器，storeValue 将源寄存器的值存储到目的寄存器中。

表 6.5 ARM 解释器寄存器安排

ARM 寄存器	Dalvik VM 使用名称	用　　途
R4	rPC	解释器中使用的程序计数器
R5	rFP	解释器中使用的栈底指针
R6	rSELF	Self 线程指针
R7	rINST	当前指令首个 16b 单元
R8	rIBASE	解释器中使用的指令基地址，用以计算 goto 地址

代码清单 6.14 dalvik/vm/compiler/codegen/arm/CodegenDriver.cpp:handleFmt11x()

```
case OP_MOVE_RESULT:
case OP_MOVE_RESULT_OBJECT: {
    /* An inlined move result is effectively no-op */
    if (mir->OptimizationFlags & MIR_INLINED)
        break;
    RegLocation rlDest=dvmCompilerGetDest(cUnit,mir,0);
    RegLocation rlSrc=LOC_DALVIK_RETURN_VAL;
    rlSrc.fp=rlDest.fp;
    storeValue(cUnit,rlDest,rlSrc);
    break;
}
```

字节码中所有要用到的寄存器在解释器栈上都有对应的存储区域，每个寄存器占用 4B。其访问方式是栈底指针＋寄存器号×4。以寄存器号为索引，比如 v5 寄存器，对应的在栈中的位置就是 FP＋20。Native Code 对数据的操作最终会反映在解释器栈上，从而使得 Native Code 和解释器之间可以平滑过渡。因此在转换过程中不论是获取 Dalvik 寄存器值还是存回运算结果，都需要有访存指令。比如对于字节码 add-int/2addr v0，v1，其转换后对应的 LIR 指令应该如代码清单 6.15 所示。

代码清单 6.15 转换后的 LIR

```
ldr     r0,[r5,#0]
ldr     r1,[r5,#4]
adds    r0,r0,r1
str     r0,[r5,#0]
```

从中可以看到，首先获取了该操作码的两个操作数，Dalvik 寄存器 v0 与 r0 对应，v1 和 r1 对应。ARM 指令 adds 实现了 add-int/2addr 字节码的功能。运算结束后，将运算结果存回 Dalvik 寄存器 v0 中。

Dalvik JIT 寄存器分配方案比较简单。其主要方案是查找寄存器池，如果有寄存器没有被分配出去，就分配该 LIR 寄存器，之后设置寄存器已经被分配。寄存器池包含的寄存器主要有{r0，r1，r2，r3，r4PC，r7，r8，r9，r10，r11，r12}。JIT 在没有优化时，有改变寄存器值的指令都应该有取操作数指令(load)和存储回内存(store 指令)。

如果基本块类型不是代码块，则需要进行特殊的处理。这其中也主要分为两类，分别是入口块及出口块和 Chaining Cell。后端对于入口块的处理非常简单，如果入口块中有代码，

则和正常的代码块一样处理,否则只生成一个标签。对于 Chaining Cell,在所有代码块转换完成后,遍历所有基本块进行处理。处理生成的代码为加载解释器中的 PC 值至 r0 寄存器,在数据区写入了跳转回的 Dalvik 字节码地址,最后跳转回解释器,交由解释器处理。

6.5.2　LIR 转换为机器码

内存中以 LIR 指令号为索引维护了一张指令映射表 EncodingMap(文件 dalvik/vm/compiler/codegen/arm/Assemble.cpp),每一条 LIR 指令都映射为其中的 ARM 指令。ArmLIR 指令操作码 kThumbAddPRR 的意思是相加两个寄存器的值并将结果存储到寄存器中,相对应地在 ARM 中为 add 指令,该指令共有 16 位,其中 15～9 位为操作码 add,8～6 位为源寄存器,5～3 位为另一个源寄存器,2～0 位为目的寄存器,指令格式如代码清单 6.16 所示。

代码清单 6.16　kThumbAddPRR 格式

```
[0001100]rm[8..6]rn[5..3]rd[2..0]
```

其在映射表对应的主要数据如表 6.6 所示。

表 6.6　kThumbMovImm 指令映射表

ARM 指令对应元素	kThumbMovImm 对应值	说　明
opcode	kThumbAddPRR	ARM 操作码
skeleton	0x1800	操作码对应的十六进制表示
k0	kFmtBitBlt	指明目的操作数有起始和终止位置
ds	2	目的操作数从第 2 位开始
de	0	目的操作数在第 0 位结束
k1	kFmtBitBlt	指明源操作数有起始和终止位置
s1s	5	第一源操作数从第 5 位开始
s1e	3	第二源操作数在第 3 位结束
s2	kFmtBitBlt	第二源操作数
s2s	8	第二源操作数从第 8 位开始
s2e	6	第二元操作数在第 6 位结束
name	Adds	指令名称
Fmt	r! 0d, #! d	输出 LIR 时的格式

映射表以 LIR 中的操作码为索引,获得映射表中的数据,操作码将会转换成十六进制。同时扫描所有操作数,并且操作数也将转换成十六进制。转换后的操作码和操作数合并,组成一条二进制指令,如图 6.13 所示。依次对每一条 LIR 进行转换,该过程在 dvmCompilerTrace 中是一个 while 循环。

转换 LIR 的过程中,需要注意的是:①在 MIR 转换为 LIR 阶段,已经规划好寄存器的使用,因而这个过程中不再进行寄存器的分配;②转换之前,需要计算每条指令的偏移地址,同时需要保证 4B 对齐。这个过程由 assembleInstructions 函数完成。

转换结束后,需要在 Translation Cache 中安装 Native Code。解释器保存了 Translation

Cache 的起始地址以及已使用大小。现在只需根据两者计算本次安装的初始地址。Dalvik JIT 调用 Linux 系统函数 memcpy 拷贝编译好的 Native Code 至相应的位置，并更新 pJitEntryTable。至此，Trace 的编译也就结束了。

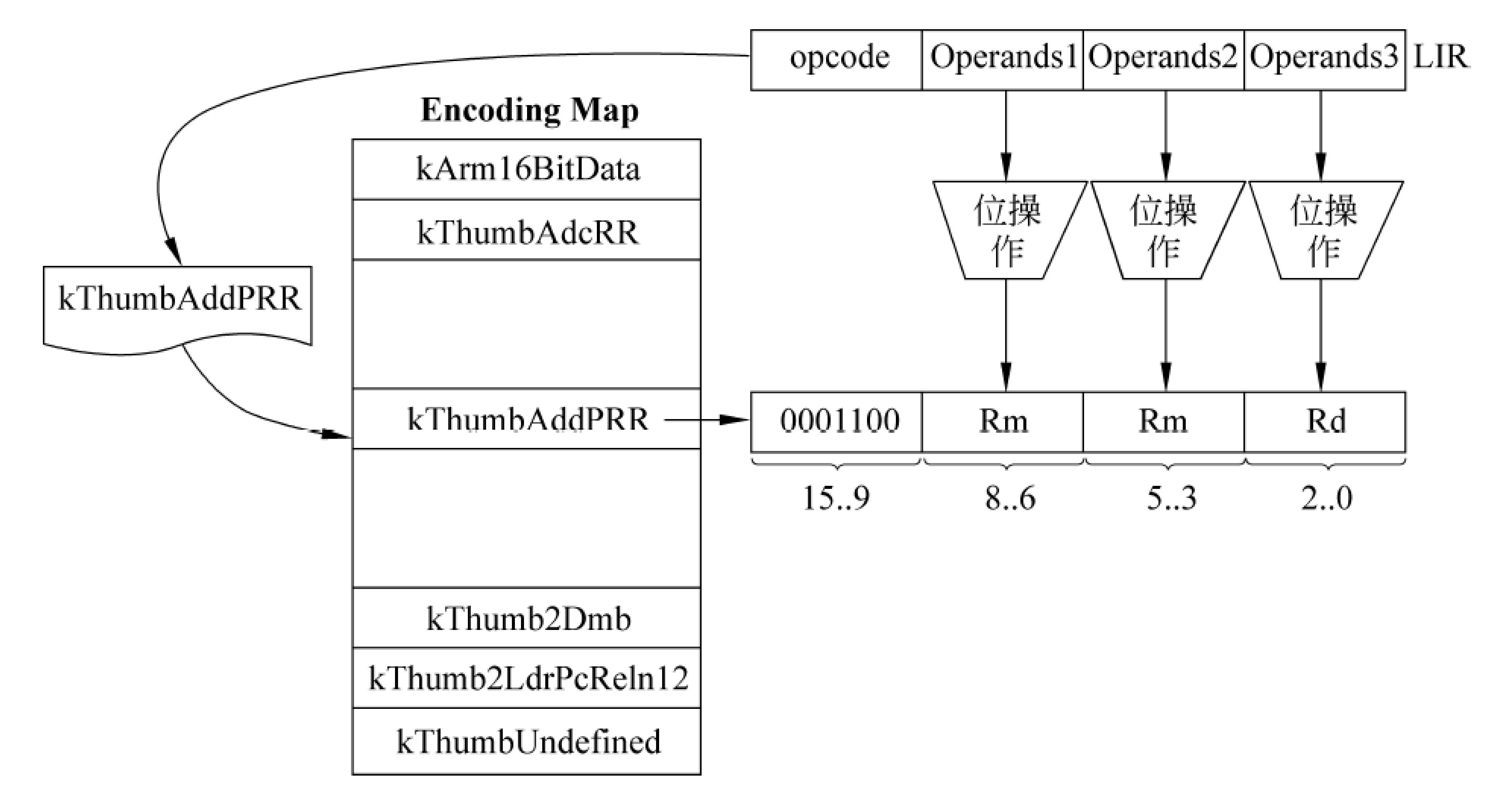


图 6.13　LIR 转换为二进制指令图

小　　结

Dalvik JIT 作为 Dalvik VM 的一个子模块，只和解释器模块有耦合，内聚性非常高。本章分为三个部分介绍 Dalvik JIT 的原理。首先从宏观上介绍了 Dalvik JIT 的功能框架及其设计原理；接着模仿传统编译器，将 Dalvik JIT 划分为前端和后端两部分，分别详细分析了其设计原理。